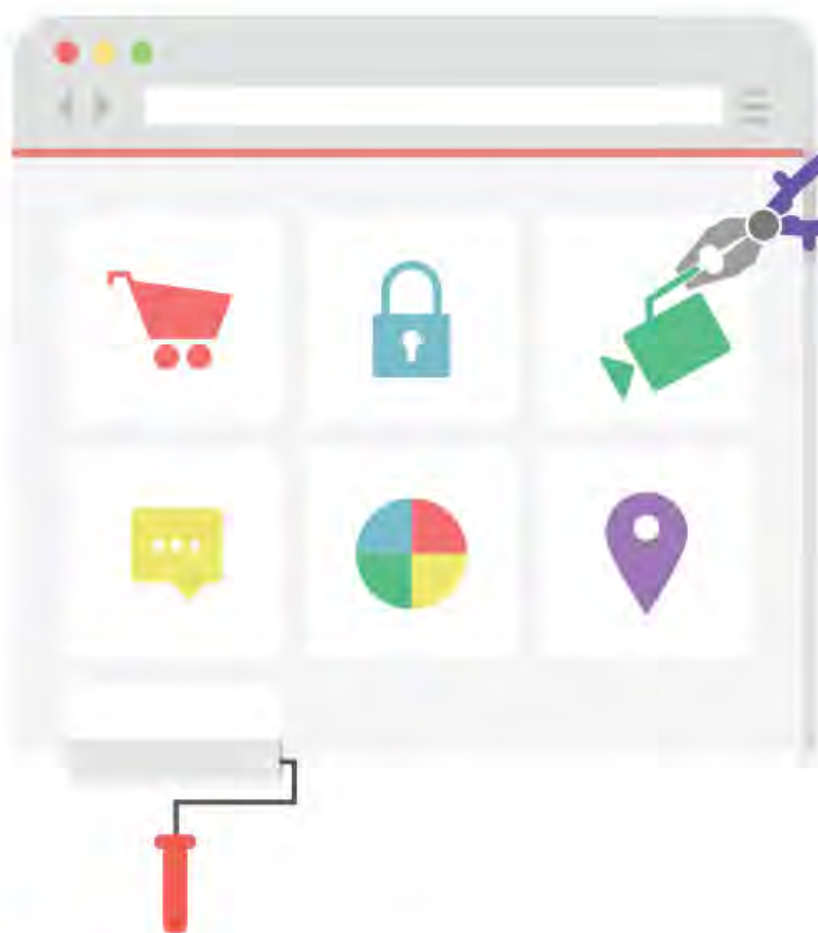
PHP e Laravel

Crie aplicações web como um verdadeiro artesão

RODRIGO TURINI

"A tudo que tenho de mais importante nessa vida, minha esposa Jordana e nossa primeira filha, Clara."
– Rodrigo Turini

Agradecimentos

Por mais que eu queira e tente muito, nenhum conjunto de palavras será o suficiente para expressar meu eterno agradecimento à minha esposa Jordana e nossa família, que agora está maior com a chegada da pequena Clara.

Gostaria também de agradecer ao Daniel Turini, que sempre influenciou em meu crescimento pessoal e profissional. Foi ele quem deu meu primeiro computador e me encaminhou ao mundo da Ciência da computação e recentemente aos diversos projetos em PHP que desenvolvemos juntos.

Agradeço também às equipes da Caelum, Alura e Casa do Código, que são empresas sensacionais constituídas por profissionais extraordinários. Em especial a meus amigos Paulo e Guilherme Silveira, Victor Harada, Maurício Aniche e Luiz Corte Real.

Prefácio

Apesar de ser intimamente ligado ao Java, uma outra linguagem de programação, há pouco mais de um ano recebi o desafio de manter a aplicação interna de uma empresa americana, totalmente escrita em PHP. No começo, o trabalho foi bem mais difícil do que eu esperava, já que se tratava de um projeto legado, nem um pouco orientado a objetos e muitíssimo complicado de manter. Depois de um tempo, vimos que a solução mais simples para esse caso seria reescrever todo o projeto, utilizando boas práticas, testes automatizados e as possibilidades mais atuais da linguagem. É aí que entra o Laravel.

Eu já havia tido algum contato com esse framework quando estava trabalhando no conteúdo técnico do curso de PHP da Caelum, mas ao estudá-lo a fundo enquanto viabilizava as opções atuais, tive a sensação de que seria uma excelente escolha. Hoje, eu tenho certeza.

O objetivo desse livro será mostrar por que o Laravel é a aposta do mercado atual e minha primeira opção de framework MVC em PHP. Criar aplicações elegantes em pouco tempo nunca foi tão fácil.

Sumário

Capítulo 1

Introdução

1.1 O projeto e as tecnologias

Nosso projeto será de controle de estoque. Como usuário final, seremos capazes de gerenciar os produtos que serão persistidos em um banco de dados MySQL, visualizar com facilidade os que estão em falta no estoque, e mais. O contexto é simples, mas será uma boa base para explorar os poderosos recursos e facilidades que o Laravel oferece.

Ao final deste livro, teremos uma listagem parecida com:

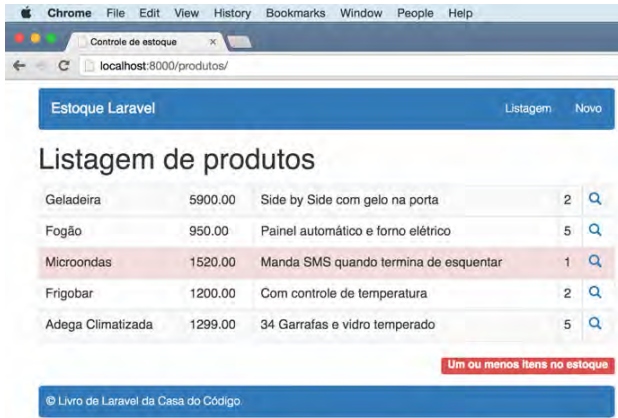



Fig. 1.1: Listagem com alguns produtos.

Adição de produtos com validação de dados:

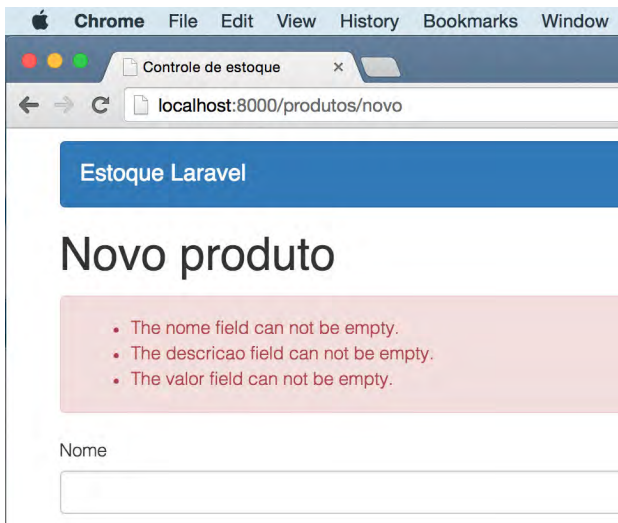
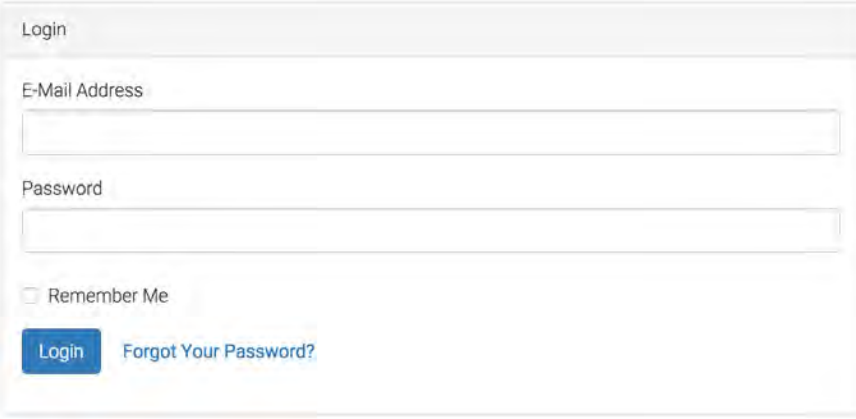



Fig. 1.2: Adição e validação de produtos.

Autenticação e segurança:

Fig. 1.3: Formulário de Login da aplicação.

E muito mais, como veremos a seguir. Um ponto importante é que o livro não focará apenas nas funcionalidades da aplicação, mas sim nos conceitos e recursos como um todo. Você entenderá, entre diversos outros, como funciona o MVC e importantes conceitos relacionados aos projetos web.

1.2 O que veremos durante o livro

Durante a leitura veremos:

- Como criar e configurar uma aplicação com Laravel.
- Como configurar rotas no arquivo `routes.php`.
- Como funciona o padrão arquitetural MVC, como ele se aplica ao framework e quais suas vantagens.
- Quais as configurações necessárias para integrar seu projeto web com um banco de dados.
- Como utilizar o facade `DB` para executar instruções no banco de dados.
- Como tirar proveito do Eloquent, um poderoso framework ORM. Suas vantagens e principais operações.

- Como enviar parâmetros para a view, redirecionar para outras ações e recuperar parâmetros da requisição do navegador.
- Como dominar o Blade, um mecanismo de template padrão do Laravel, utilizar layouts e operadores lógicos.
- Serializar o resultado em diferentes formatos, como JSON.
- Criar validações de dados e entender o conceito de `Form Requests`.
- Cuidar da autenticação e segurança com `Middlewares`.
- Como tirar proveito dos zilhares de comandos do Artisan.

1.3 Download do Laravel

Se ainda não tem o Laravel instalado, você pode fazer seu download e ver um passo a passo detalhado de instalação em seu próprio site:

http://laravel.com/docs/installation

Se tiver qualquer problema no processo de instalação, que vai variar de acordo com o sistema operacional, não desanime. Envie um e-mail na lista de discussões desse livro que vamos ajudá-lo. O endereço é:

https://groups.google.com/d/forum/livro-laravel

Laravel Homestead

Se preferir, em vez de instalar o PHP, Laravel e todas as suas dependências em sua máquina local, você pode utilizar o *Laravel Homestead* para preparar o ambiente de uma forma bem simples e elegante em uma máquina virtual. O *Homestead* é uma solução oficial e já inclui PHP 5.6, MySQL, além de diversos outros recursos de que você pode precisar para desenvolver aplicações completas em Laravel. Se quiser, você pode ler mais a respeito e ver instrução de instalação e uso em:

http://laravel.com/docs/homestead

1.4 Acesse o código desse livro

O código completo do projeto que desenvolveremos no decorrer desse livro está disponível em meu repositório do GitHub. Você pode acessá-lo em:

https://github.com/Turini/estoque-laravel

1.5 Aproveitando ao máximo o conteúdo

Para tirar um maior proveito dessa leitura, não fique preso à teoria. Você pode e deve acompanhar o livro com seu editor favorito aberto, escrevendo todo o código e testes dos capítulos. Além disso, eu sempre recomendo que você vá além escrevendo novos testes para solidificar ainda mais o conhecimento.

1.6 Tirando suas dúvidas

Ficou com alguma dúvida? Não deixe de me enviar um e-mail. A lista de discussão a seguir foi criada exclusivamente para este livro:

https://groups.google.com/d/forum/livro-laravel

Essa lista é um canal de comunicação direta comigo e com os demais leitores, portanto fique à vontade para levantar discussões técnicas, apontar correções, indicar melhorias etc. Seu feedback é sempre muito bem-vindo.

Além da lista, não deixe de consultar a documentação do framework durante todo o aprendizado. Ela é bem completa e explicativa:

http://laravel.com/docs/

Outro recurso que você pode utilizar para esclarecer suas dúvidas e participar ativamente na comunidade é o fórum do GUJ. Lá você não só pode perguntar, mas também responder, editar, comentar e assistir a diversas discussões sobre o universo da programação.

http://www.guj.com.br/

Capítulo 2

Novo projeto com Laravel

2.1 Como criar cinco telas de uma aplicação web?

Imagine uma aplicação que tenha as funções de adicionar, remover, listar, enviar e-mail, entre diversas outras que são essenciais para toda aplicação web. Para cada uma, devemos executar um código de lógica, buscar ou atualizar informações do banco, mostrar um HTML como resposta. Bastante coisa repetitiva, não é? Será que alguém não pode nos ajudar?

2.2 Framework, pra que te quero?

Independente da linguagem ou tecnologia que estamos usando, um conceito global é: **não queremos ficar nos preocupando com infraestrutura**. É aí

que os frameworks entram. Eles nos ajudam e muito a agilizar o processo de desenvolvimento, de forma organizada, evitando repetições de código e muito mais.

Quem nunca foi criar um projeto novo e acabou copiando a estrutura de algum outro projeto que já tinha criado antes? Isso acontece porque boa parte dessa estrutura será igual, você não precisa reinventar a roda a cada novo projeto. Essa é uma das ideias dos frameworks, oferecer essa estrutura padrão entre os projetos, de forma bem organizada e fácil de manter, segundo as melhores práticas do mercado. Essa reutilização de código entre vários projetos vai lhe poupar muito tempo e trabalho. Precisa conectar com o banco? Enviar um e-mail? Migrar seu banco de dados? Você perceberá que o *Laravel*, assim como diversos outros frameworks do mercado, já tem tudo isso pronto e pré-configurado.

Ao longo desse livro você perceberá que não precisa usar frameworks, não é obrigatório. Mas mesmo assim você não vai querer mais viver sem eles, que nos tornam muito mais produtivos e simplificam bastante o processo de desenvolvimento.

Algumas outras opções famosas

Além do *Laravel*, que vamos aprender, existem diversas opções bastante interessantes no mercado. Uma das mais populares é o *Zend Framework 2*, da própria **Zend Technologies**. Além dele, há também o *CodeIgniter*, *Symphony*, *CakePHP*, *Phalcon*, entre diversos outros.

O *Laravel* é uma das maiores apostas da atualidade. Muito se deve à sua simplicidade, sintaxe, flexibilidade e rica documentação. Além de seu site oficial, que é o http://laravel.com, você também conta com bastante conteúdo, discussões, exemplos de código, perguntas e respostas em um site de receitas, o http://laravel-recipes.com/. Eu recomendo que, além do livro, você use e abuse desses sites para dominar completamente a ferramenta.

2.3 Novo projeto: Uma app de controle de estoque

Para começar a aprender Laravel, vamos criar um novo projeto chamado `estoque`. Fazer isso é verdadeiramente simples, desde o começo você já percebe as vantagens em usar o framework. Quer ver como é fácil? Se você já tem o Laravel instalado, tudo que precisa fazer é rodar o comando `laravel new` pelo terminal, passando o nome do projeto que queremos criar. Em nosso caso será:

```
laravel new estoque
```

A saída será parecida com:

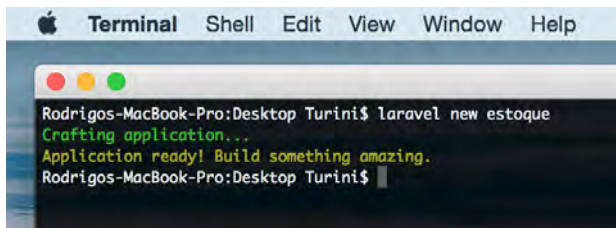



Fig. 2.1: Criando o projeto via terminal, na pasta Desktop.

O texto *Application ready! Build something amazing* será exibido e pronto, ele cuidou de todo o trabalho pesado. Note que uma pasta com o nome do projeto (`estoque`, neste caso) foi criada no mesmo diretório em que você executou o comando. Ela já tem toda a estrutura de pastas, algumas classes e dependências configuradas.

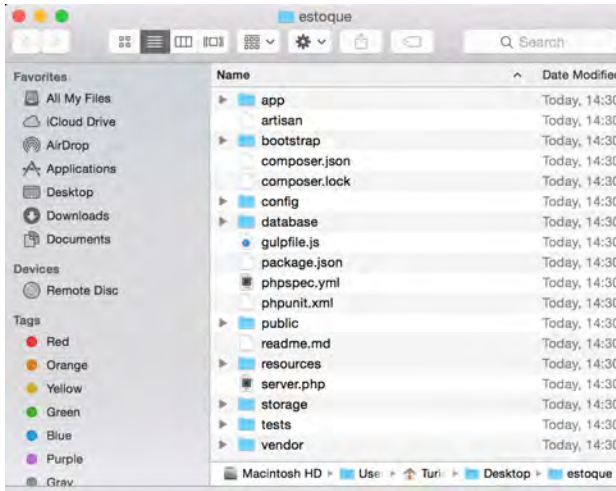



Fig. 2.2: Estrutura inicial do projeto.

Legal, não é? Se você está se perguntando o que significa cada uma dessas pastas, não se preocupe: em breve vamos falar mais sobre essa estrutura inicial e a entenderemos mais a fundo no decorrer do livro. Logo você estará dominando tudo isso. Mas por agora, vamos rodar o projeto `estoque` para garantir que tudo está funcionando conforme o esperado?

Normalmente usamos o próprio *Apache* do servidor em que fizermos *deploy*, mas em ambiente de desenvolvimento, podemos servir nossa aplicação utilizando o comando `php artisan serve`, que executa a aplicação no servidor de desenvolvimento do PHP. Basta rodar esse comando de dentro da pasta do projeto:

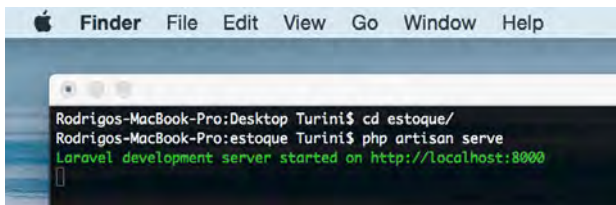



Fig. 2.3: Executando servidor de desenvolvimento do PHP com Laravel.

Navegando nas pastas do OS pelo terminal

Você pode utilizar o comando `cd` (*change directory*) para navegar pelas suas pastas via terminal. Um exemplo em Windows seria:

```
D:\> cd Desktop\estoque
D:\Desktop\estoque>
```

O mesmo comando pode ser utilizado em um ambiente Unix (Linux ou Mac OS). Para listar, há uma diferença. Em Windows utilizamos o comando `dir`:

```
D:\Desktop\estoque> dir
    // vai mostrar todos os arquivos
```

Porém, nos outros sistemas que foram citados o comando será `ls`. Repare:

```
turini ~ $ cd Desktop/estoque
turini/Desktop/estoque ~ $ ls
    // vai mostrar todos os arquivos
```

Considerando, portanto, que meu projeto se chama `estoque` e foi criado dentro da pasta `Desktop`, para executar o comando que inicia o servidor você precisará executar os dois comandos:

```
cd Desktop/estoque/
php artisan serve
```

Se tudo correu bem, a mensagem `Laravel development server started on` http://localhost:8000 deve ter aparecido no seu terminal. Vamos testar? Basta acessar essa URL no navegador de sua preferência. O resultado será:

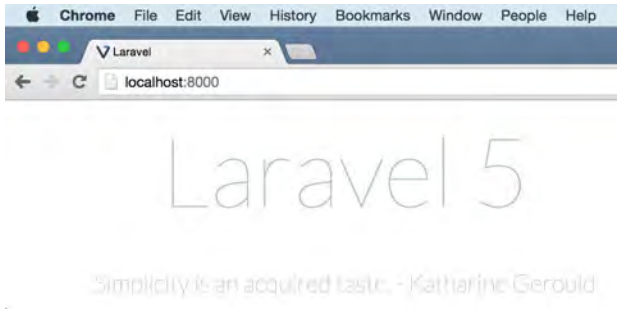



Fig. 2.4: Página de boas-vindas do Laravel 5.

Note que, para criar um projeto e executar o Laravel, nós não precisamos de nenhuma configuração extra. Basta criar o projeto em alguns segundos, com o comando `laravel new`, e sair usando! Isso é muito conveniente quando estamos começando um novo projeto. Podemos gastar nosso tempo com o que realmente importa: nossas regras de negócio.

Precisa fazer alguma configuração adicional? O framework faz todo o possível para que você não precise ficar configurando nada, mas em alguns momentos isso pode ser necessário. Se algo de que você precisa não está configurado por *default*, como por exemplo o `locale`, você pode fazer isso facilmente pelo arquivo de configurações presente em `app/config/app.php`.

Quer saber mais sobre essas configurações adicionais? Talvez você queira dar uma olhada na página de configurações do framework, disponível em:

http://laravel.com/docs/configuration

Mas, por enquanto, não precisaremos de nenhuma configuração adicional em nosso projeto. Se ele executou sem nenhum problema, já estamos prontos para prosseguir.

2.4 Entendendo a estrutura de pastas

Como vimos, ao criar um novo projeto, diversas classes e arquivos foram criados. O objetivo dessa estrutura inicial é oferecer um padrão e o mínimo de esforço possível para começar o seu projeto. Isso é bem legal, mas claro, caso preferir, você também pode renomear as classes e mudar a estrutura de pastas

para ficar como melhor lhe agrada, o framework não impõe muitas restrições quanto a isso.

Durante todo o livro conheceremos um pouco mais sobre as pastas e essa estrutura inicial, mas desde já podemos ter uma breve noção do que vai em cada lugar. Essas são algumas das principais pastas:

- **app**: nela ficam seus **m**odelos, **v**iews e **c**ontrollers, que serão bem detalhados no próximo capítulo. Em poucas palavras, é onde boa parte do seu código vai ficar. Ela possui uma série de subdiretórios, como `Commands`, `Console`, `Http`, `Events`, entre outros. Não se preocupe em entender o significado de cada um deles agora, vamos vê-los melhor conforme formos precisando.

- **config**: como o nome já indica, é onde ficam os arquivos de configuração do seu projeto. Se você precisar alterar as configurações de cache, e-mail, banco de dados, entre outras, já sabe onde encontrar.

- **public**: é a pasta pra onde seu web server vai apontar. Lá fica o arquivo `index.php`, que aponta para sua aplicação. Além disso, é comum colocarmos os arquivos `css`, `imagens`, `javascript` e todos os demais arquivos públicos nesse diretório.

- **vendor**: é onde fica o *source code* do Laravel, plugins e outras dependências. Tudo que você usar de terceiros (bibliotecas, frameworks etc.) deve ficar nela.

Esse é só um pouco, claro, mas já é o bastante por agora. Vamos entrar mais a fundo nesse conteúdo no decorrer do livro, mas se quiser adiantar, talvez queira dar uma olhada na página do Laravel que explica essa estrutura inicial:

http://laravel.com/docs/structure

2.5 Alterando o namespace padrão com Artisan

O *namespace* padrão de toda aplicação com Laravel é `App`, mas é muito comum e bastante recomendado que você altere o namespace para o nome da

sua aplicação. Como fazer isso? É muito fácil, basta rodar um simples comando e pronto.

Pra subir o server e testar, usamos o `php artisan serve`, lembra? Esse **Artisan** é uma ferramenta de linha de comando já inclusa no framework. Ela nos oferece uma série de comandos úteis para tornar nosso desenvolvimento mais produtivo.

Para mudar o *namespace*, por exemplo, podemos usar o `php artisan app:name`. Vamos mudá-lo para **estoque**, que é o nome do projeto. Basta executar o seguinte comando pelo terminal, dentro da pasta de seu projeto:

```
php artisan app:name estoque
```

A saída será parecida com:

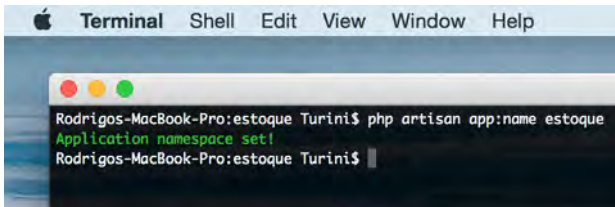



Fig. 2.5: Alterando o namespace com Artisan.

Tudo pronto, *namespace* alterado!

2.6 Criando nossa primeira lógica

Agora que já conhecemos um pouco mais sobre o Laravel, queremos ensiná-lo como queremos que ele reaja quando alguém acessar determinada URL, isto é, criar as nossas próprias rotas. Mas quão complicado é fazer isso?

Quando acessamos http://localhost:8000/, ou seja, a URL / da nossa aplicação, em algum lugar foi configurado que a página padrão do Laravel deveria ser exibida, não é? Esse trabalho é feito no o arquivo `routes.php`, que fica dentro da pasta `app/Http/`. Abra o arquivo para conferir seu conteúdo, que deve estar parecido com:

```
<?php

// comentário omitido

Route::get('/', 'WelcomeController@index');

Route::get('home', 'HomeController@index');

Route::controllers([
    'auth' => 'Auth\AuthController',
    'password' => 'Auth\PasswordController',
]);
```

Isso pode mudar de acordo com a versão do Laravel que você estiver utilizando. Na versão 5.1, por exemplo, o conteúdo será:

```
<?php

// comentário omitido

Route::get('/', function () {
    return view('welcome');
});
```

Como você pode ver, ele já vem com algumas rotas configuradas, como a `/` que nos leva para a página *default* do Laravel. O código pode parecer diferente no começo, mas não é complicado. Vamos entender em detalhes cada linha de código desse arquivo, mas antes disso, apague todo esse código para criarmos nossa própria rota. Que acha? É bem fácil, basta deixar seu arquivo vazio, só com a tag do php:

```
<?php
    // nosso código vai aqui
```

E agora, usar o `Route:get` para definir uma nova rota. Podemos fazer algo como:

```
<?php
```

```
Route::get('/', function()
{
    return 'Primeira lógica com Laravel';
});
```

Sucesso, já criamos nossa primeira rota. Vamos entender o que aconteceu?

Usamos o método estático `get`, da classe `Route`, passando dois parâmetros. O primeiro é a rota (caminho, ou *path* como comumente é chamado) que será acessado pelo navegador. O segundo parâmetro é uma função com a resposta que deverá ser enviada. Em poucas palavras, esse código ensina que, quando alguém acessar a URL `/`, o Laravel deve retornar o texto `Primeira lógica com Laravel` para o navegador.

Vamos testar? Basta rodar o comando `php artisan serve` para subir o servidor novamente (caso ainda não esteja startado) e acesse http://localhost:8000/ em seu navegador. Veja a resposta:

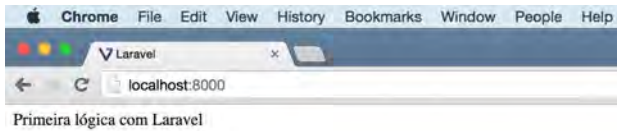



Fig. 2.6: Primeira lógica com Laravel.

Perfeito, o texto foi exibido no navegador! Mandamos um texto comum, mas podemos responder com qualquer HTML válido. Por exemplo, envolvendo esse texto em um `h1` para ter um destaque maior na página:

```
<?php

Route::get('/', function()
{
    return '<h1>Primeira lógica com Laravel</h1>';
});
```

Você não precisa restartar o servidor, basta acessar a página novamente para ver o resultado:

Fig. 2.7: Resposta com tag html.

2.7 Conflito entre rotas do Laravel

É importante perceber desde já que você pode criar quantas rotas quiser no arquivo `routes.php`:

```
<?php

Route::get('/', function()
{
    return '<h1>Primeira lógica com Laravel</h1>';
});

Route::get('/outra', function()
{
    return '<h1>Outra lógica com Laravel</h1>';
});
```

Agora temos duas rotas, uma para a URL `/` e outra para `/outra`. Mas o que aconteceria se as duas fossem registradas na mesma URL? Na `/`, por exemplo.

```
<?php

Route::get('/', function()
{
    return '<h1>Primeira lógica com Laravel</h1>';
```

```
});

Route::get('/', function()
{
    return '<h1>Outra lógica com Laravel</h1>';
});
```

Tente rodar esse código para ver o resultado, a segunda rota vai sobrescrever a primeira e o texto `Outra lógica com Laravel` será exibido. Ou seja, em caso de ambiguidade sempre a última rota é quem será registrada. Há outras formas de lidar com ambiguidade, como quando usamos diferentes métodos HTTP, mas entraremos nesse assunto um pouco mais à frente.

2.8 Antes de continuar

Se quiser praticar um pouco mais, eu recomendo que crie outras rotas e novos testes antes de seguir para o próximo capítulo. Não se preocupe em entender as partes que ainda não vimos, ok? Foque apenas neste conteúdo inicial. Ficou com qualquer dúvida? Não deixe de perguntar! Lembre-se que um grupo de discussões foi criado especialmente para este livro:

https://groups.google.com/d/forum/livro-laravel

Agora que já sabemos o essencial, podemos partir para as regras de negócio do nosso sistema de estoque. Está preparado?

Capítulo 3

MVC e conexão com banco de dados

3.1 Model-View-Controller

Repare novamente na forma como fizemos para registrar nossa primeira lógica:

```
Route::get('/', function()
{
    return '<h1>Primeira lógica com Laravel</h1>';
});
```

O problema de definir as rotas dessa forma, com o código de resposta implementado diretamente em uma **função anônima**, é que não estamos se-

guindo nem um pouco as boas práticas da orientação a objetos. O código fica todo espalhado, difícil de manter e reutilizar.

Conforme nossa aplicação for crescendo, vamos precisar fazer acesso ao banco de dados, executar lógicas com nossas regras de negócio etc. Tudo isso ficará dentro do mesmo arquivo `route.php`? Seria uma bagunça! Além disso, vamos misturar nossa lógica de negócio com nossa lógica de apresentação, o HTML de resposta.

Precisamos de uma forma melhor de dividir as responsabilidades de nossa aplicação. É aí que entra o **MVC**, ou **M**odel **V**iew **C**ontroller.

A grande ideia desse padrão arquitetural é que você separe suas regras de negócio em 3 camadas, cada uma com sua responsabilidade muito bem definida:

- **Model** é a camada onde ficam nossas regras de negócio, nossas entidades e classes de acesso ao banco de dados.
- **View** é a responsável por apresentar as páginas e outros tipos de resultado para o usuário (ou mesmo para outros sistemas, que se comunicam). É a resposta que o framework envia para o navegador, que normalmente é um HTML.
- **Controller** é quem cuida de receber as requisições web e decidir o que fazer com elas. Nessa camada definimos quais **model**os devem ser executados para determinada ação e para qual **view** vamos encaminhar a resposta. Em outras palavras, essa camada quem faz o link entre todas as outras.

Diversos frameworks, das mais diferentes linguagens, seguem esse padrão do MVC. Com Laravel não é diferente, o fluxo fica assim:

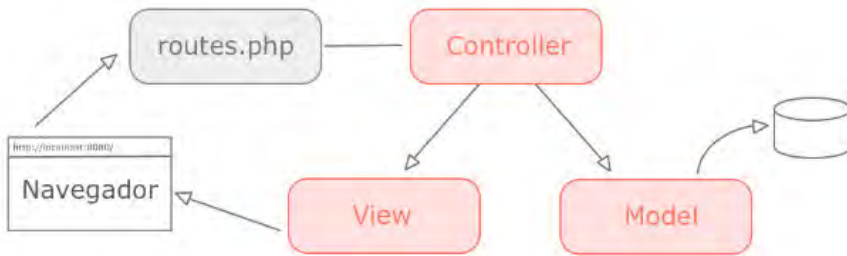



Fig. 3.1: Fluxo do MVC com Laravel.

Repare que, quando nosso cliente envia uma requisição pelo navegador, primeiramente temos um arquivo PHP, que é o `routes.php`, que está frente de todos. Ele cuida de atender as requisições e enviá-las para o local correto, no caso os nossos controllers. Os controllers, por sua vez, decidem o que fazer com as requisições, passando pela camada de model (que fazem acesso ao banco, executam as regras de negócio etc.) e logo em seguida delegam pra *view* que será exibida como resposta no navegador do cliente.

Agora que já sabemos um pouco da teoria, vamos colocar o MVC em prática?

3.2 Colocando MVC em prática

Em vez de definir todas as lógicas do nosso sistema nesse arquivo único, o `routes.php`, vamos organizá-las desde o início em *Controllers* distintos. Nosso sistema de estoques vai ter uma página principal, com a listagem de produtos. Podemos começar por ela.

Vamos criar um novo arquivo chamado `ProdutoController`, dentro da pasta `app/Http/Controllers`, que é o diretório padrão para esse tipo de classe. Dentro do controller, crie um método chamado `lista`. O arquivo `ProdutoController.php` ficará assim:

```
<?php namespace estoque\Http\Controllers;

class ProdutoController {
```

```
    public function lista(){
        // nosso código vai aqui
    }
}
```

Note que, como estamos trabalhando com uma estrutura de diretórios, tivemos que definir o *namespace* no cabeçalho do arquivo. Outra regra importante é que todo controller do Laravel deve herdar de uma classe chamada `Controller`, que foi criada automaticamente junto com nosso projeto. Para isso, só precisamos adicionar `extends Controller` na declaração da nossa classe, que ficará assim:

```
<?php
class ProdutoController extends Controller {

    public function lista(){
        // nosso código vai aqui
    }
}
```

A classe `Controller` existe exclusivamente para que seja possível definirmos lógicas em comum que podem ser compartilhadas entre todos os controller de nossa aplicação. Além disso, ela já traz alguns *imports* (`uses`) essenciais para os controllers, que serão detalhados conforme formos precisando.

Inicialmente, vamos fazer o método `lista` retornar um HTML puro, com o cabeçalho da listagem de produtos. O código ficará assim:

```
<?php
class ProdutoController extends Controller {

    public function lista(){
        return '<h1>Listagem de produtos com Laravel</h1>';
    }
}
```

E agora que temos esse comportamento definido no controller, vamos criar uma rota para ele no arquivo `routes.php`, que continua responsá-

vel pelo registro de nossas rotas e outras configurações, mas a diferença é que agora ele apenas apontará para o método do controller que deve ser executado.

O arquivo de rotas deve ficar assim:

```
<?php

Route::get('/produtos', 'ProdutoController@lista');

// demais rotas omitidas
```

Note que o padrão é o nome do controller, seguido de um `@` e o nome do método. Agora quando uma requisição for feita para a URL `/produtos`, o método `lista` do `ProdutoController` será executado. Bem simples, não acha?

Vamos testar? Basta acessar http://localhost:8000/produtos em seu navegador.

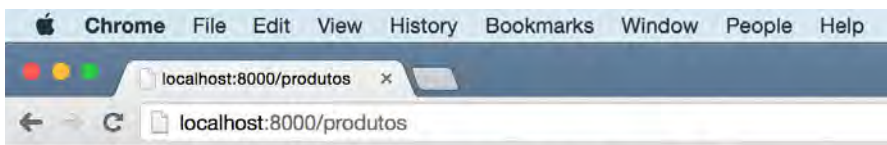



Fig. 3.2: Início da listagem de produtos.

Sucesso, o HTML de resposta foi exibido conforme esperado.

3.3 Trabalhando com banco de dados

A listagem ainda está muito simples, ainda estamos mostrando apenas um texto onde queremos mostrar todos os produtos. Nosso objetivo agora será

buscar essas informações de um banco de dados, como é natural nas aplicações do dia a dia.

Iniciando com MySQL

Usaremos o banco de dados MySQL em nosso projeto, já que é um dos bancos de dados relacionais mais utilizados no mercado, é gratuito e bastante simples de instalar. Se você ainda não tem o MySQL instalado, pode fazer download pelo link:

http://dev.mysql.com/downloads/mysql/

Fig. 3.3: Página de download do MySQL.

Ao final da página, você encontrará um campo para escolher seu sistema operacional. Selecione e depois clique em download, na opção que preferir:

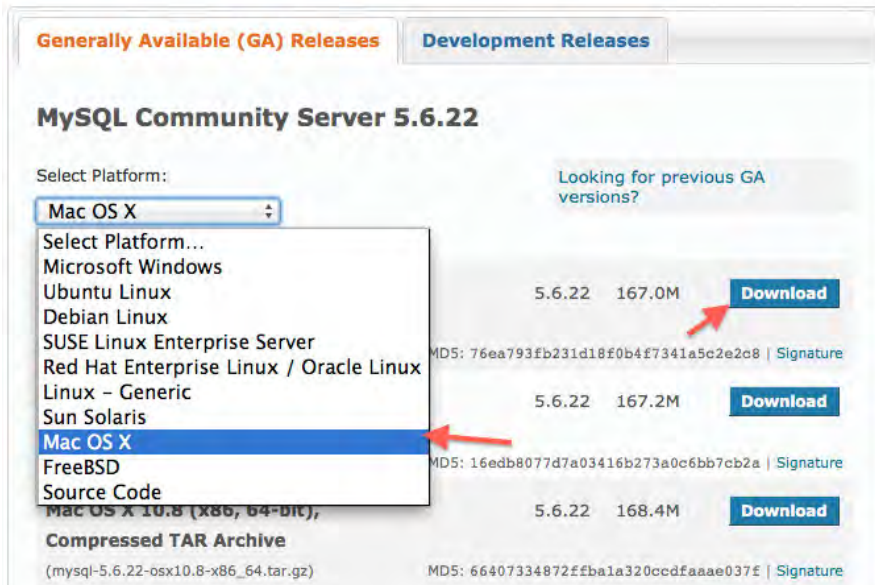
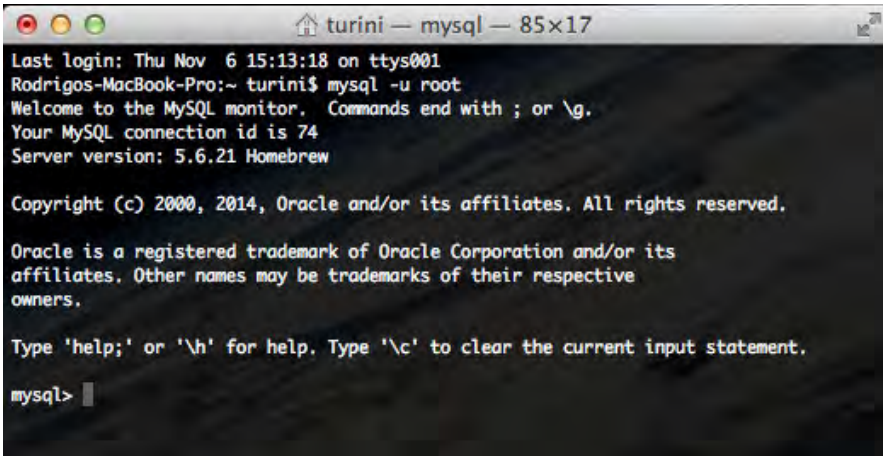



Fig. 3.4: Selecionando opção de download de acordo com o sistema operacional.

No mesmo site, você encontra um tutorial de instalação de acordo com o seu sistema operacional. Ou você pode acessar o tutorial diretamente em:

http://dev.mysql.com/doc/refman/5.7/en/installing.html

Após baixar e instalar o MySQL, vamos nos logar no MySQL e criar a base de dados (*database*) de nosso projeto. O primeiro passo é bem simples, tudo que você precisa fazer para se logar é abrir seu terminal e digitar:

```
mysql -u SEU_USUARIO -p SUA_SENHA
```

Como em meu caso o usuário é `root` e eu não tenho uma senha, só preciso fazer:

```
mysql -u root
```

```
turini — mysql — 85×17
Last login: Thu Nov  6 15:13:18 on ttys001
Rodrigos-MacBook-Pro:~ turini$ mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 74
Server version: 5.6.21 Homebrew

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Fig. 3.5: Tela inicial do MySQL Command Line

MySQL Command Line ou Workbench?

Há quem prefira utilizar o *MySQL Workbench*, que é uma ferramenta visual. Pela simplicidade, nos exemplos do livro vamos utilizar o *MySQL Command Line*. Nele, nós executamos as instruções diretamente pela linha de comando, no terminal.

Se tiver qualquer dificuldade para instalar ou executar os comandos, não pense duas vezes antes de mandar um e-mail para lista deste livro.

Agora que já estamos logados, podemos criar uma nova *database* chamada `estoque_laravel`. Isso pode ser feito com o comando:

```
create database estoque_laravel;
```

Após executá-lo, a saída deve ser parecida com:

```
Query OK, 1 row affected (0.00 sec)
```

Perfeito, já temos o banco. Para evitar que você tenha que criar todas as tabelas e cadastrar algumas informações manualmente nesse momento, vamos importar o arquivo `dump.sql` com alguns produtos já cadastrados em

uma tabela de produtos. Não se preocupe, muito em breve substituiremos esse *dump* com recursos oferecidos pelo Laravel. Um arquivo de *dump* é um arquivo de texto com instruções SQL, com os *inserts* de dados ou até mesmo com as instruções de criação de tabelas.

Criando seu próprio dump

Se quiser saber mais sobre esse arquivo de *dump*, ou mesmo criar o seu próprio, você pode dar uma olhada no link:

http://dev.mysql.com/doc/refman/5.0/en/mysqldump-sql-format.html

Isso pode ser bem útil quando queremos fazer *backups* de segurança de nossas bases de dados.

Faça o download do `dump.sql` nesse link:

http://bit.ly/1BKZMgo

Feito isso, tudo que precisamos fazer para importar esse *dump* em nosso banco de dados é executar a seguinte instrução no terminal:

```
mysql -uroot estoque_laravel < CAMINHO_COMPLETO_PARA_O_DUMP
```

Para facilitar o processo, você pode copiar o arquivo `dump.sql` para a pasta raiz de seu usuário, e executar apenas:

```
mysql -uroot estoque_laravel < dump.sql
```

Ótimo! Já podemos dar o próximo passo.

3.4 Configuração e conexão com MySQL

Agora que já temos o banco instalado e configurado, nosso objetivo será estabelecer uma conexão com o MySQL. O Laravel torna essa tarefa bem fácil, basta adicionar as informações do nosso banco no arquivo `config/database.php`. Ele já vem pré-configurado no projeto, só precisamos mudar o trecho de código do MySQL, que deve estar assim:

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'forge'),
    'username'  => env('DB_USERNAME', 'forge'),
    'password'  => env('DB_PASSWORD', ''),
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
],
```

No campo database usaremos `estoque_laravel`, usuário `root` e senha vazia. Caso seu MySQL tenha um usuário ou senha diferente, basta substituir o valor desses campos. O trecho de configuração do MySQL no arquivo `database.php` ficará assim:

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'estoque_laravel'),
    'username'  => env('DB_USERNAME', 'root'),
    'password'  => env('DB_PASSWORD', ''),
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
],
```

Pronto, isso é tudo que precisamos configurar para acesso ao banco de dados. Vamos testar? Uma forma simples, porém bem manual, de rodar SQLs pelo Laravel é utilizando a classe `DB`, presente em `Illuminate\Support\Facades\DB`. Ela tem uma série de métodos para nos ajudar a realizar operações no banco de dados, como `select`, `insert` etc.

Quer ver como é simples? Para buscar todos os registros da tabela produtos, basta utilizar o método `select` com a SQL de consulta:

```
$produtos = DB::select('select * from produtos');
```

Nosso controller ficará assim:

```
<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;

class ProdutoController extends Controller {

    public function lista(){

        $produtos = DB::select('select * from produtos');

        return '<h1>Listagem de produtos com Laravel</h1>';
    }
}
```

Todos os valores dos produtos do banco de dados serão retornados em um *array*, que chamamos de `$produtos`. Vamos garantir que isso está funcionando? Podemos mudar o método `lista` para *concatenar* o nome e descrição de cada um dos produtos na string de resposta. O código ficará assim:

```
<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;

class ProdutoController extends Controller {

    public function lista(){

        $html = '<h1>Listagem de produtos com Laravel</h1>';

        $html .= '<ul>';

        $produtos = DB::select('select * from produtos');

        foreach ($produtos as $p) {
            $html .= '<li> Nome: '. $p->nome .',
                      Descrição: '. $p->descricao .'</li>';
        }
```

```
        $html .= '</ul>';

        return $html;
    }
}
```

Achou o código feio? Então somos dois! Mas não se preocupe com isso, logo ele será melhorado, o nosso objetivo agora é garantir que a conexão com o banco de dados foi estabelecida corretamente. Como já temos alguns produtos cadastrados pelo `dump`, ao acessar http://localhost:8000/produtos o resultado deverá ser parecido com:

Fig. 3.6: Lista de produtos com registros do banco.

Excelente, com quase nenhuma configuração já conseguimos nos conectar e executar consultas no banco de dados!

3.5 Para saber mais: Environment

Você provavelmente percebeu que adicionamos as configurações de banco de dados como segundo parâmetro de um método `env`, diretamente no arquivo de `database.php`.

```
'database'  => env('DB_DATABASE', 'estoque_laravel'),
'username'  => env('DB_USERNAME', 'root'),
'password'  => env('DB_PASSWORD', ''),
```

Mas quando o projeto estiver pronto, não vamos querer usar esses dados em produção. Por uma questão de segurança, é importantíssimo que o banco de produção tenha uma senha, e que ela fique protegida. Além disso, não queremos que os desenvolvedores usem o banco de dados de produção para fazerem seus testes futuros. É aí que entra o método `env`.

O Laravel possui um recurso conhecido como *Environment*, que nos possibilita definir um conjunto de configurações de podem mudar de acordo com o ambiente de execução, como o caso do banco de dados.

Veja que seu projeto já possui um arquivo `.env.example`, presente no diretório raiz. Dentro dele você encontrará, entre outras opções, as linhas:

```
DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

O método `env`, utilizado no arquivo `database.php`, primeiro procura pela configuração definida no arquivo de *environment* e, caso não encontre, utiliza o segundo valor que passamos. Em outras palavras, se quisermos que o Laravel passe a utilizar as configurações de produção em vez dos valores fixos passados para o arquivo database, tudo o que precisamos fazer é criar um arquivo `.env` com essas configurações. Se você estiver utilizando a versão mais nova do Laravel, ele já existirá. Basta adicionar o conteúdo:

```
DB_HOST=meuprojeto.com.br
DB_DATABASE=estoque_producao
DB_USERNAME=production
DB_PASSWORD=uma_senha_super_hiper_secreta
```

Mas cuidado. Esse arquivo terá informações importantes de seu ambiente de produção, como a senha do banco de dados. É bastante recomendado que você não deixe esse arquivo no projeto ou *commit* no repositório, caso esteja utilizando alguma ferramenta de controle de versão como o git. Se quiser, você pode ver mais detalhes sobre essa funcionalidade em:

http://laravel.com/docs/configuration#environment-configuration

3.6 Aonde chegamos e para onde queremos ir

Com um pouco de teoria e prática, o significado do padrão MVC começa a fazer sentido. Como vimos, em vez de deixar todas as responsabilidades no arquivo de rotas, cada comportamento pode ficar muito bem *encapsulado* em seu devido controller. As vantagens são inúmeras, mas pra mim as principais são:

- **Legibilidade:** pois em vez de ter várias funções anônimas e inúmeras linhas de código em um mesmo arquivo, tudo fica muito bem distribuído. Cada comportamento em seu devido lugar, em classes e métodos com nomes bem definidos.
- **Manutenibilidade:** pois se tudo está bem organizado e encapsulado, não precisaremos mexer em um arquivo com 1500 linhas sempre que um problema aparecer. A listagem de produtos parou de funcionar? Sei que tenho que ir a um método do `ProdutoController`. Não consigo adicionar usuários? Sei que tenho que verificar na classe `UsuarioController`.

Viu só? Isso nada mais é do que bom uso da orientação a objetos, que cada vez mais faz parte e influencia o PHP e seus frameworks.

Mas espera, nosso código ainda não obedece o princípio das 3 camadas do MVC, por enquanto todo o trabalho está sendo feito na camada controller! Isso não é bom, vamos resolver?

Capítulo 4

Camada de modelo, migrations e seeds

Até então usamos um *dump* do banco de dados para importar a tabela produtos e alguns dados prontos, mas, ao criar um novo projeto, nem sempre teremos essas informações prontas, não é? Neste capítulo veremos como o Laravel pode ajudar nesse trabalho, criando todas essas informações do zero, sem termos que nos preocupar em acessar o banco de dados e executar esses *create table*s e *insert*s na mão.

4.1 O modelo Produto

Nosso primeiro passo será criar uma classe de **model**o que **represente nossa tabela de produtos** do banco de dados. Algo bem simples, como:

```
<?php namespace estoque;

use Illuminate\Database\Eloquent\Model;

class Produto extends Model {

}
```

Veja que nossa classe `Produto` precisa herdar de `Model`, mas não se preocupe com isso por enquanto, em breve entenderemos o motivo e ganhos dessa herança.

Mas espera, acha trabalhoso digitar essas linhas pra criar a classe na mão? Que tal deixar o **Artisan** fazer isso pra você? Isso mesmo, o **Artisan** faz muito mais do que subir o servidor, ele tem uma série de recursos que exploraremos no decorrer do livro. Um deles é o *make:model*, que criará toda a estrutura da classe de modelo para você. Repare como é simples:

```
php artisan make:model Produto
```

Após executar esse comando, a classe será criada automaticamente:

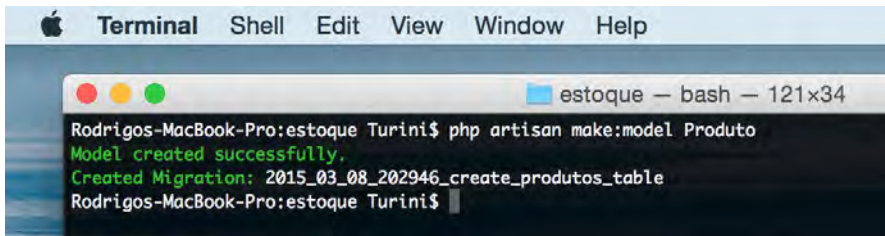



Fig. 4.1: Comando do Artisan para criação de modelos.

Tudo pronto, procure pela classe `Produto` dentro da pasta `app` de seu projeto, ela estará assim:

```
<?php namespace estoque;

use Illuminate\Database\Eloquent\Model;
```

```
class Produto extends Model {
    //
}
```

Legal, não é? Até nisso o Laravel nos ajuda. O comando `php artisan make:model` é opcional, claro, mas muito útil em nosso dia a dia. Você só precisa passar o nome da classe e todo o código padrão será criado para você, assim em nenhum momento precisaremos nos preocupar em lembrar de herdar de `Model`, adicionar o *import* e *namespace* dos nossos modelos.

A partir de agora, a classe `Produto` representará a tabela *produtos* que será criada no banco de dados. Mas em qual lugar isso foi configurado? A resposta é: em nenhum. Quando não definimos explicitamente, o framework assume que a tabela terá o nome da classe com letra minúscula e no plural. Nesse caso, a classe `Produto` será vinculada à tabela `produtos`.

Mas se você preferir, ou mesmo precisar, também é possível configurar isso explicitamente. Basta adicionar em nosso modelo uma propriedade com visibilidade `protected`, chamada `table`:

```
<?php namespace estoque;

use Illuminate\Database\Eloquent\Model;

class Produto extends Model {

    protected $table = 'produtos';
}
```

4.2 Trabalhando com migrations

Se você estiver utilizando a versão 5.0 do Laravel, além da classe `Produto`, um arquivo de migração (*migration*) chamado `2015_05_25_151007_create_produto_table` foi criado. A partir da versão 5.1, ele não é criado automaticamente; logo, você precisará executar o comando com o argumento `-m`:

```
php artisan make:model Produto -m
```

Esse arquivo será o responsável por ensinar ao Laravel como criar, atualizar ou recuperar o estado anterior do esquema de seu banco de dados. *Migrations*, ou migrações, trabalham como um controle de versão do seu banco de dados. Veja que, inclusive, o nome do arquivo de migração começa com a data em que ela foi criada, pois assim o Laravel saberá exatamente a ordem em que elas devem ser executadas.

Ao abrir esse arquivo de migração, que por padrão será criado na pasta `database/migrations`, veremos que uma classe `CreateProdutosTable` foi criada com o seguinte conteúdo:

```
<?php

use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateProdutosTable extends Migration {

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('produtos', function(Blueprint $table)
        {
            $table->increments('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
```

```
        Schema::drop('produtos');
    }
}
```

Essa classe, que herda de `Migration`, terá dois métodos importantíssimos. O primeiro (`up`) ensina como criar a tabela produto, enquanto o segundo (`down`) mostra como desfazê-la, ou seja, fazer um *rollback*.

Porém, repare que, no método `up`, apenas o *id* e *timestamp* da tabela são definidos. Queremos que, além disso, um produto tenha *nome*, *valor*, *descricao* e *quantidade*, portanto adicionaremos essas as informações a seguir:

```
public function up()
{
    Schema::create('produtos', function(Blueprint $table)
    {
        $table->increments('id');
        $table->string('nome');
        $table->decimal('valor', 5, 2);
        $table->string('descricao');
        $table->integer('quantidade');
        $table->timestamps();
    });
}
```

Utilizamos o método `string` para os campos de texto, `integer` para a quantidade e `decimal` para o valor do produto. Existem diversas outras opções, como `date` para datas, `time` para hora, entre outros. Você pode ver uma relação completa dos métodos desse *builder* em:

http://laravel.com/docs/schema#adding-columns

Com isso, nossa migração está pronta, o método `up` fornece todas as informações necessárias para que o Laravel saiba criar a tabela produtos, e o método `down` ensina ao framework como desfazer essa alteração, que nesse caso será apagando a tabela:

```
public function down()
{
    Schema::drop('produtos');
}
```

4.3 Apagando a tabela produtos

Agora que já temos uma migração para fazer esse trabalho, vamos apagar a tabela de produtos já existente em nosso banco de dados. Para isso, basta acessar o *MySQL* e executar a instrução:

```
DROP TABLE produtos;
```

Tudo pronto. Acessar a listagem de produtos pelo navegador agora resultaria em um erro, claro. Para que o código volte a funcionar precisamos executar a migração de criação da tabela. Mas como?

4.4 Executando migrações

Executar uma *migration* é uma tarefa extremamente simples. O comando `php artisan migrate` é quem fará todo o trabalho. Ao executá-lo, teremos como resultado algo como:

```
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2015_05_25_151007_create_produtos_table
```

Veja que, além de nossa migração de produtos, uma migração de criação de usuários e reset de senha foi executada. Essas migrações foram criadas automaticamente junto com o projeto, para a funcionalidade de autenticação que veremos mais à frente.

Vamos conferir o resultado? Basta acessar o banco de dados do projeto e conferir a estrutura da tabela que foi criada. Podemos fazer isso com o comando `desc` do *MySQL*.

```
mysql> desc produtos;
```

O resultado será:

```
mysql> desc produtos;*: lista todas as rotas registradas. Este comando é especialment
+------------+------------------+------+-----+---------------------+----------------+
| Field      | Type             | Null | Key | Default             | Extra          |
+------------+------------------+------+-----+---------------------+----------------+
| id         | int(10) unsigned | NO   | PRI | NULL                | auto_increment |
| nome       | varchar(255)     | NO   |     | NULL                |                |
| valor      | decimal(5,2)     | NO   |     | NULL                |                |
| descricao  | varchar(255)     | NO   |     | NULL                |                |
| quantidade | int(11)          | NO   |     | NULL                |                |
| created_at | timestamp        | NO   |     | 0000-00-00 00:00:00 |                |
| updated_at | timestamp        | NO   |     | 0000-00-00 00:00:00 |                |
+------------+------------------+------+-----+---------------------+----------------+
```

Fig. 4.2: Estrutura da tabela produtos no banco de dados.

Excelente, já temos a tabela produtos com tudo que precisamos.

E quanto ao rollback?

Eventualmente surgirá a necessidade de desfazer uma migração, seja por um erro de digitação em algum de seus campos, adicionar alguma informação ou algo do tipo. Se você acabou de executar o comando `migrate` e viu que alguma coisa não saiu como esperado, basta executar um `php artisan migrate:rollback` que a ultima migração será desfeita. Legal, não é? Mas cuidado. Muitas vezes será mais interessante criar uma nova migração com as mudanças, em vez de sair desfazendo as últimas migrações executadas. Lembre-se que, caso alguma informação já tenha sido inserida na tabela, ao fazer *rollback* ela será perdida.

Você pode criar quantas migrações quiser com o comando:

```
php artisan make:migration NOME_DA_MIGRATION
```

4.5 Inserindo dados na tabela produtos

O código voltará a funcionar, porém, ao acessar a listagem de produtos novamente pelo navegador, nenhum dado será exibido. É o esperado, afinal, apagamos a tabela de produtos anterior e criamos uma nova.

Em breve implementaremos a função de adicionar novos produtos, mas enquanto isso podemos inserir alguns produtos manualmente no banco de dados. Uma forma simples de fazer isso seria acessando o *MySQL* e executando diretamente os *insert*s, mas veremos uma forma ainda mais interessante e automatizada, com um novo recurso do Laravel chamado `Seed`.

A ideia é simples, um *seeder* é uma classe que popula seu banco de dados com algumas informações iniciais normalmente informações de testes. Toda a mágica acontece dentro de uma classe chamada `DatabaseSeeder`, presente no diretório `database/seeds`. Quando fizemos um novo projeto, essa classe foi criada com o seguinte conteúdo:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder {

    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        Model::unguard();

        //$this->call('UserTableSeeder');
    }
}
```

Note que há uma linha comentada com o exemplo de como executar um *seeder*, com auxílio do método `call`.

Criando o ProdutoTableSeeder

Vamos agora criar um *seeder* que deverá inserir alguns dados na tabela produtos. A princípio o código pode parecer estranho, mas é

verdadeiramente simples. Assim como o `DatabaseSeeder`, nosso `ProdutoTableSeeder` deve herdar da classe `Seeder` e implementar seu método `run`, como a seguir:

```
class ProdutoTableSeeder extends Seeder {

    public function run()
    {
        // código vai aqui
    }
}
```

Nada demais, não é? Como por padrão o `DatabaseSeeder` não vem com *namespace*, é uma prática muito comum definirmos os *seeder*s dentro de um mesmo arquivo. Portanto, o código ficará assim:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder {

    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        Model::unguard();

        //$this->call('UserTableSeeder');
    }
}

class ProdutoTableSeeder extends Seeder {

    public function run()
```

```
    {
    // código vai aqui
    }
}
```

Ótimo, agora só falta adicionarmos alguns dados! Assim como usamos o `DB:select` para fazer um `select` no banco de dados, podemos usar um `DB:insert` para fazer uma inserção. O código ficará assim:

```
class ProdutoTableSeeder extends Seeder {

  public function run()
  {

    DB::insert('insert into produtos
      (nome, quantidade, valor, descricao)
      values (?,?,?,?)',
      array('Geladeira', 2, 5900.00,
      'Side by Side com gelo na porta'));

    DB::insert('insert into produtos
      (nome, quantidade, valor, descricao)
      values (?,?,?,?)',
      array('Fogão', 5, 950.00,
      'Painel automático e forno elétrico'));

    DB::insert('insert into produtos
      (nome, quantidade, valor, descricao)
      values (?,?,?,?)',
      array('Microondas', 1, 1520.00,
      'Manda SMS quando termina de esquentar'));
  }
}
```

Não se preocupe em entender o `DB:insert` por enquanto, voltaremos a falar sobre esse método muito em breve. O que importa nesse momento é que, em vez de executarmos 3 inserts no banco de dados manualmente, deixamos esse código isolado em uma classe responsável por fazer esse trabalho para nós. O ganho é que, em qualquer ambiente, podemos executar esse *seed*

para inserir alguns dados na tabela de produtos sempre que quisermos. Legal, não é? Só faltam dois detalhes para que o código fique completo. O primeiro é adicionar o *import* da classe `DB`, assim como fizemos no controller de produtos:

```
use Illuminate\Support\Facades\DB;
```

O outro detalhe será adicionar a chamada do `ProdutoTableSeeder` dentro do método `run` da classe `DatabaseSeeder`. O código completo ficará assim:

```
<?php

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\DB;

class DatabaseSeeder extends Seeder {

    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        Model::unguard();

        $this->call('ProdutoTableSeeder');
    }
}

class ProdutoTableSeeder extends Seeder {

  public function run()
  {

    DB::insert('insert into produtos
      (nome, quantidade, valor, descricao)
```

```
      values (?,?,?,?)',
      array('Geladeira', 2, 5900.00,
      'Side by Side com gelo na porta'));

    DB::insert('insert into produtos
      (nome, quantidade, valor, descricao)
      values (?,?,?,?)',
      array('Fogão', 5, 950.00,
      'Painel automático e forno elétrico'));

    DB::insert('insert into produtos
      (nome, quantidade, valor, descricao)
      values (?,?,?,?)',
      array('Microondas', 1, 1520.00,
      'Manda SMS quando termina de esquentar'));
  }
}
```

Excelente. Vamos testar? Podemos utilizar o comando `db:seed` do **Artisan** sempre que quisermos executar o `DatabaseSeeder`. Basta acessar o terminal e executar:

```
php artisan db:seed
```

Tudo pronto! A mensagem `Seeded: ProdutoTableSeeder` será impressa e os produtos estarão salvos no banco de dados. Acesse a listagem novamente para conferir.

Capítulo 5

Trabalhando com a View

Sabemos que nosso controller está com muitas responsabilidades, mas por onde começar a melhorá-lo? Atualmente, o código está assim:

```
<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;

class ProdutoController extends Controller {

    public function lista(){

        $html = '<h1>Listagem de produtos com Laravel</h1>';

        $html .= '<ul>';
```

```
        $produtos = DB::select('select * from produtos');

        foreach ($produtos as $p) {
            $html .= '<li> Nome: '. $p->nome .',
                      Descrição: '. $p->descricao .'</li>';
        }

        $html .= '</ul>';

        return $html;
    }
}
```

Isso não é legal por uma infinidade de motivos, por exemplo, como faríamos para estilizar esse HTML? E se eu quisesse adicionar uma imagem, uma tabela, o header, título etc., tudo isso fica mesmo dentro do código do meu controller?

5.1 Cada coisa em seu lugar: camada View

Em vez de retornar um HTML, o controller deveria delegar esse trabalho para a camada de **view**, que é a especialista do MVC nesse assunto. A mudança será simples, basta criar um arquivo PHP com toda essa lógica de visualização. Podemos chamá-lo de `listagem.php`, e é importante que ele esteja no diretório `resources/views`.

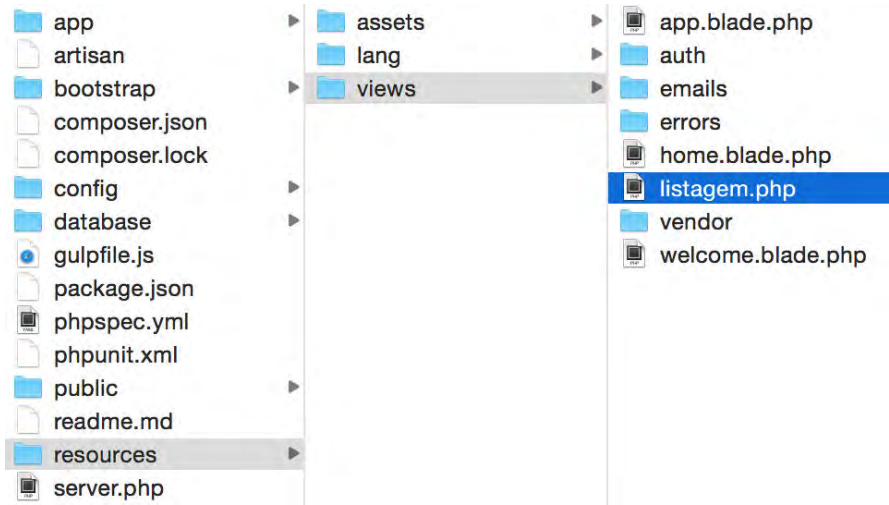



Fig. 5.1: Arquivo /estoque/resources/views/listagem.php

Nesse arquivo podemos fazer algo muito parecido com o que estamos fazendo no controller, um `foreach` em todos os produtos, mostrando seu nome, descrição, e agora que estamos no lugar certo, também podemos exibir os outros campos, que são o valor e a quantidade. Em vez de uma lista, podemos usar uma tabela para organizar um pouco melhor os dados. O código poderá ficar parecido com:

```
<html>
  <body>
    <h1>Listagem de produtos</h1>
    <table>
      <?php foreach ($produtos as $p): ?>
      <tr>
        <td><?= $p->nome ?></td>
        <td><?= $p->valor ?></td>
        <td><?= $p->descricao ?></td>
        <td><?= $p->quantidade ?></td>
      </tr>
      <?php endforeach ?>
    </table>
```

```
  </body>
</html>
```

Agora que temos o arquivo pronto, só precisamos ensinar ao controller que, em vez de escrever o HTML todo, ele deve simplesmente delegar o trabalho para essa nossa view. Mas como fazer isso?

Vamos por partes, antes de mais nada, vamos tirar todo aquele código HTML do método `lista`. Ele ficar assim:

```
class ProdutoController extends Controller {

    public function lista(){

        $produtos = DB::select('select * from produtos');

        return // o que retornar aqui?
    }
}
```

Ufa, bem melhor, não acha? Tiramos toda a lógica de visualização de nosso controller, já que temos uma camada própria para fazer esse trabalho, a **view**. O próximo passo é ensinar ao controller que, após fazer seu trabalho, ele deve renderizar o HTML da view `listagem.php` que acabamos de criar. Para fazer isso, basta mudar o retorno do método para `view('listagem')`, como a seguir:

```
class ProdutoController extends Controller {

    public function lista(){

        $produtos = DB::select('select * from produtos');

        return view('listagem');
    }
}
```

Esse método recebe como parâmetro o nome da página que você quer exibir, sem a extensão, e para utilizá-lo você não precisa fazer nenhum *import*, já que isso é feito pela classe `Controller` que herdamos.

Vamos ver se tudo está funcionando? Basta acessar http://localhost:8000/produtos novamente. Diferente do esperado, o resultado será:

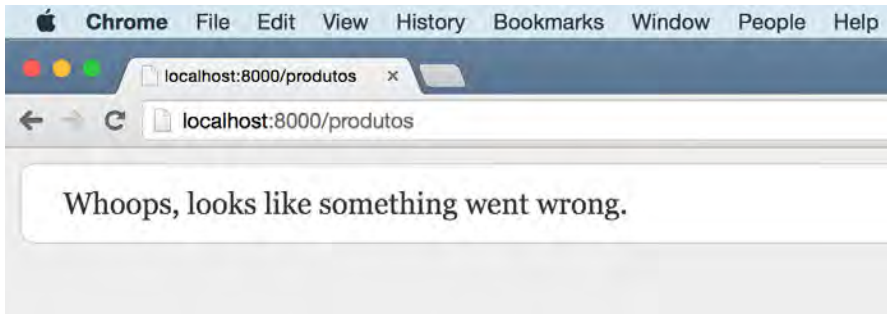



Fig. 5.2: Página de listagem com mensagem de erro.

Ops, alguma coisa deu errado, mas como descobrir? A mensagem do navegador não nos diz muito.

5.2 Consultando os arquivos de log do framework

Sempre que você receber uma mensagem como essa, dizendo que alguma coisa deu errado em seu código, você pode e deve consultar os arquivos de log da sua aplicação para ter uma descrição mais detalhada sobre o que está acontecendo. Esses arquivos ficam no diretório `/storage/logs/` e são nomeados por data.

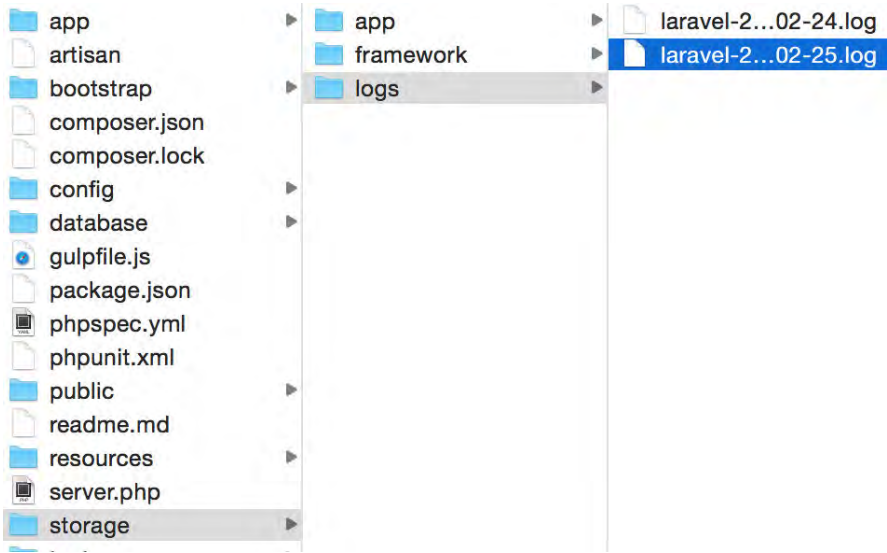



Fig. 5.3: Arquivo de log da aplicação na data atual.

Ao abrir o log da data e hora atual, podemos ver logo no início do arquivo a seguinte mensagem de erro:

```
[2015-02-25 22:07:50] production.ERROR:
    exception 'ErrorException' with message
    'Undefined variable: produtos' in
    /Users/Turini/Desktop/estoque/resources
    /views/listagem.php:9
```

Veja a mensagem: `Undefined variable: produtos`. Faz todo sentido, não é? Afinal, se criamos a variável `produtos` no controller, não podemos acessá-la na view, que é um outro arquivo.

Há ainda uma forma mais interessante de consultar os *errors* da nossa aplicação quando estamos em ambiente de desenvolvimento, sem precisar a todo o momento procurar a mensagem nos arquivos de log. É fácil, basta criar um arquivo chamado `.env` dentro da pasta do projeto, e dentro dele colocar as instruções:

```
APP_DEBUG=true
```

O seu arquivo pode ficar assim:

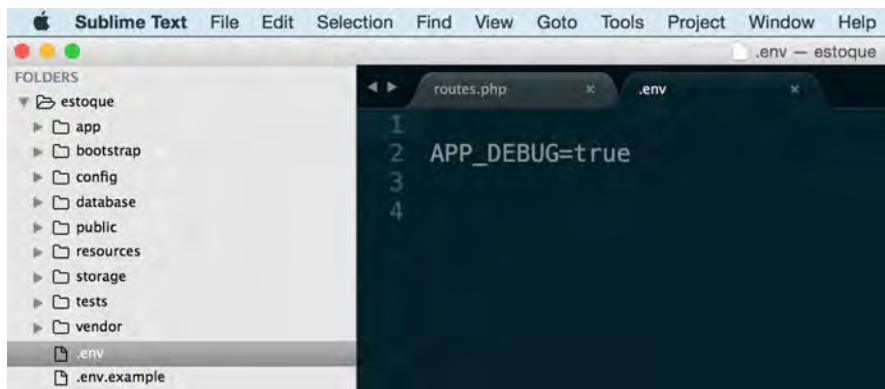



Fig. 5.4: Arquivo .env com debug ativado.

Com isso, estamos informando ao Laravel que queremos depurar as mensagens de erro, o que é bastante recomendado quando estamos desenvolvendo a aplicação. Faça a alteração e acesse a página novamente para ver o resultado:

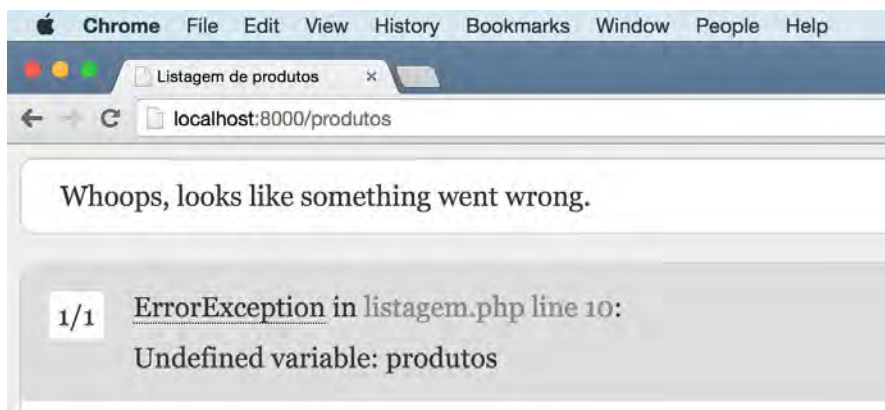



Fig. 5.5: Listagem com erro em modo debug.

Excelente! Assim fica muito mais fácil identificar o que aconteceu de errado, não é?

5.3 Incluindo parâmetros na view

O que faltou foi, de alguma forma, mandar esse *array* de produtos para a view. Poderíamos, sim, criar o array diretamente no arquivo `listagem.php` da view, mas não estaríamos respeitando a divisão de responsabilidades do MVC. A view é responsável pela lógica de visualização, não pelo acesso ao banco de dados nem nada desse tipo.

Uma forma muito mais interessante de fazer isso é, no retorno do controller, ensinar ao Laravel que queremos deixar essa variável `produtos` acessível pela view com auxilio do método `with`:

```
$produtos = DB::select('select * from produtos');
view('listagem')->with('produtos', $produtos);
```

Veja que passamos para o método `with` uma chave e valor, ou seja, a forma como ele será acessado na *view* (**produtos**, nesse caso) e o valor associado a ele. O código completo fica assim:

```
<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;
use estoque\Produto;

class ProdutoController extends Controller {

    public function lista(){

        $produtos = DB::select('select * from produtos');

        return view('listagem')->with('produtos', $produtos);
    }
}
```

Já podemos testar. Acessando a listagem novamente teremos como resultado:

Fig. 5.6: Listagem de produtos.

Pronto, tudo continua funcionando e agora nosso código está muito mais bem organizado. O controller faz seu trabalho e ao final delega a responsabilidade de apresentação do resultado para a view, que é a camada que cuida disso. Bem justo, não é?

5.4 Alguns detalhes e curiosidades

Esse método *view*, que usamos no retorno do nosso controller, é um *helper method* para simplificar esse processo de trabalhar com views. Se você já viu algum código com as versões anteriores do Laravel, vai se lembrar que para fazer o mesmo trabalho tínhamos que escrever:

```
View::make('listagem')->with('produtos', $produtos);
```

No fim das contas, o que acontece é que, ao chamar o método *view*, sem passar nenhum parâmetro, ele vai retornar uma implementação do contrato *View* definido pelo Laravel. Chamando o método com parâmetro, ele delegará a chamada para o método `make`, assim como era feito anteriormente só que com uma sintaxe um pouco mais enxuta e agradável.

Incluindo parâmetros com magic methods

Uma curiosidade é que, em vez de escrever:

```
view('listagem')->with('produtos', $produtos);
```

Você pode chamar um método `withProdutos`:

```
view('listagem')->withProdutos($produtos);
```

Faça a alteração para testar, o resultado será o mesmo! Como isso é possível?

Na verdade, esse método "não existe". Em nenhum momento os desenvolvedores escreveram um método `withProdutos` dentro da `view`, mas sim um *magic method* que faz esse trabalho dinamicamente.

Em outras palavras, se você chamou `withAlgumaCoisa`, ele usará o `algumaCoisa` como chave do parâmetro disponibilizado na view.

Passando um array de dados para a view

Existem diversas outras formas de passar os dados para a view, além do método `with`. Uma das mais conhecidas e utilizadas é passando um array como segundo parâmetro do método `view`. Em vez de fazer:

```
view('listagem')->with('produtos', $produtos);
```

Você faria algo como:

```
return view('listagem', ['produtos' => $produtos]);
```

Ou extraindo o array para uma variável:

```
$data = ['produtos' => $produtos];
return view('listagem', $data);
```

Ou ainda criando o array e adicionando cada item manualmente:

```
$data = [];
$data['produtos'] = $produtos;
return view('listagem', $data);
```

O que vai mudar é a sintaxe, mas o efeito será o mesmo. Você pode escolher a forma que preferir.

Mais alguns métodos: exists e file

Outro ponto é que você pode verificar a existência de uma view com o método `exists`:

```
if (view()->exists('listagem'))
{
    return view('listagem');
}
```

Ou mesmo usar o método `file` para gerar a view a partir de um caminho/diretório diferente:

```
view()->file('/caminho/para/sua/view');
```

Legal, não é? Se quiser conhecer um pouco mais sobre a view, eu recomendo que dê uma boa olhada em sua documentação. O link é:

http://laravel.com/docs/views

5.5 Melhorando a aparência da nossa listagem

Agora que nosso HTML está no lugar certo, que acha de melhorar um pouco o visual da listagem? O Laravel já traz um arquivo chamado `app.css` com alguns estilos prontos, para ajudá-lo a poupar tempo com esse trabalho também. Esse arquivo usa internamente o conhecido Bootstrap (http://getbootstrap.com) para o estilo, Glyphicons (http://glyphicons.com/) para os ícones e muito mais.

Não encontrou esse arquivo de CSS?

Não deixe de verificar se o arquivo `app.css` está presente no diretório `/public/css` do seu projeto. Se não estiver, significa que você está utilizando Laravel na versão 5.1 ou maior. Desde o 5.1, por uma decisão interna dos desenvolvedores do framework, esse e alguns outros arquivos foram deixados como opcionais. Para adicioná-los em seu projeto, basta instalar o pacote `scafold`: https://github.com/bestmomo/scafold.

Se preferir, você também pode copiar os arquivos do projeto que desenvolvemos durante esse livro, no repositório: https://github.com/Turini/estoque-laravel.

Para começar a usar esse estilo, basta adicionar o `head` no seu HTML e o `link` com o caminho do arquivo, algo como:

```
<html>
<head>
    <link href="/css/app.css" rel="stylesheet">
    <title>Controle de estoque</title>
</head>
<body>
<!-- continuação do seu html aqui -->
```

Note que já aproveitamos para colocar um título na página também.

Ao acessar a listagem você já deve perceber alguma diferença, mas por enquanto, só na fonte:

Fig. 5.7: Listagem de produtos com css e título.

Ainda queremos estilizar a tabela, mas como? É bem simples, basta adicionar a classe `table` que você já vai ter algum resultado. O HTML da tabela deve ficar assim:

```
<table class="table">
  <?php foreach ($produtos as $p): ?>
    <tr>
        <!-- código omitido -->
    </tr>
  <?php endforeach ?>
</table>
```

Essa simples mudança já deixa nossa listagem com um visual bem mais agradável:

Fig. 5.8: Utilizando classes do bootstrap na tabela.

Além da classe `table`, também podemos usar:

- **table-striped** para zebrar as linhas
- **table-bordered** para adicionar bordas
- **table-hover** para efeito de hover

Há diversos outros, que você pode encontrar na documentação do bootstrap:

http://getbootstrap.com/css/#tables

Vamos usar alguns desses estilos? Basta adicionar as novas classes na tabela:

```
<table class="table table-striped table-bordered table-hover">
<!-- restante do html omitido -->
```

Você pode usar qualquer outro estilo que preferir.

Além disso, para desgrudar a tabela e o título das bordas da página, podemos envolvê-las em uma `div` com a classe **container**. O arquivo `listagem.php` completo pode ficar assim:

```
<html>
<head>
    <link href="/css/app.css" rel="stylesheet">
    <title>Controle de estoque</title>
</head>
<body>
  <div class="container">
    <h1>Listagem de produtos</h1>
    <table class="table table-striped
           table-bordered table-hover">
      <?php foreach ($produtos as $p): ?>
      <tr>
        <td><?= $p->nome ?></td>
        <td><?= $p->valor ?></td>
        <td><?= $p->descricao ?></td>
        <td><?= $p->quantidade ?></td>
      </tr>
      <?php endforeach ?>
    </table>
  </div>
</body>
</html>
```

O resultado final será:

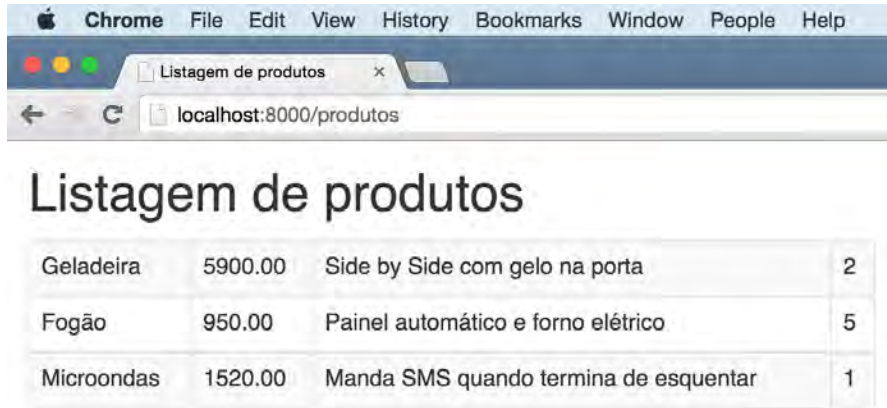



Fig. 5.9: Aparência final da listagem de produtos.

5.6 Aonde chegamos e para onde queremos ir

Neste capítulo demos um grande passo, que foi separar a lógica de apresentação das regras de negócio do controller. Vimos que os ganhos são muitos, já que agora cada coisa está em seu efetivo lugar.

Além disso, conhecemos o *helper method* `view` e vimos sobre as diferentes variações para inclusão de parâmetros utilizando, ou não, o método `with`. Pronto para dar mais alguns passos?

Capítulo 6

Parâmetros da request e URL

Agora que temos nosso código um pouco mais organizado, com a lógica de acesso ao banco no **controller** separada da lógica de apresentação na **view**, podemos adicionar um link em cada produto da tabela para que seja possível visualizar seus detalhes em uma nova tela.

6.1 Exibindo detalhes do produto

A mudança no HTML do arquivo `listagem.php` será pequena, só precisamos adicionar uma nova coluna com um link:

```
<h1>Listagem de produtos</h1>
<table class="table ...">
  <?php foreach ($produtos as $p): ?>
  <tr>
    <td><?= $p->nome ?></td>
```

```
    <!-- outras colunas omitidas -->
    <td>
     <a href="/produtos/mostra>">
       Visualizar
     </a>
    </td>
  </tr>
  <?php endforeach ?>
</table>
```

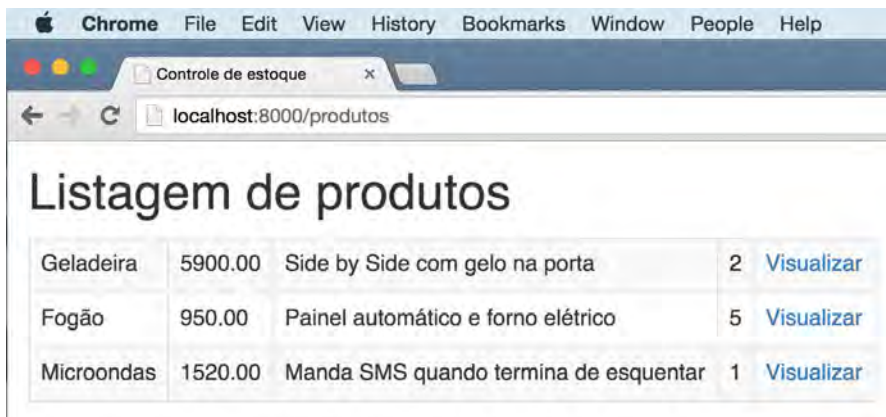



Fig. 6.1: Listagem com link de visualizar detalhes.

Ou, para deixar a aparência um pouco mais interessante, podemos usar um ícone de lupa no lugar desse texto. Como estamos usando *bootstrap*, que já vem com o framework, só precisamos adicionar um *span* com a classe do ícone que queremos utilizar. Veja como é simples:

```
<a href="/produtos/mostra">
  <span class="glyphicon glyphicon-search"></span>
</a>
```

Pronto, com isso nossa listagem ficou assim:

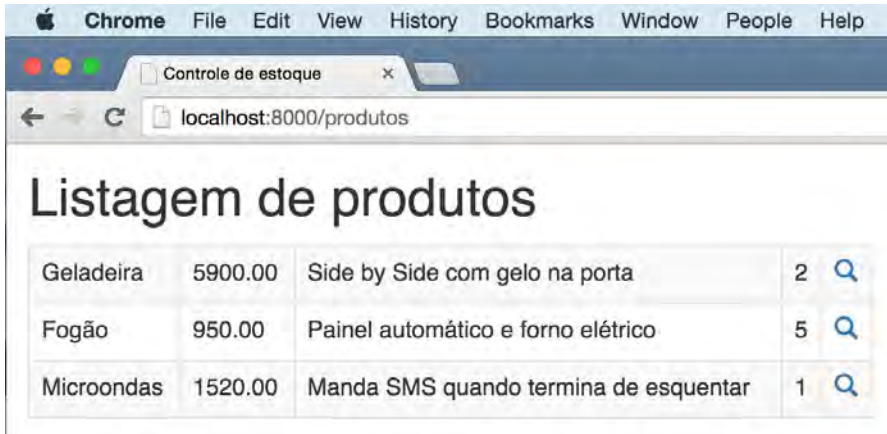



Fig. 6.2: Listagem com ícone de lupa no link.

Criando uma nova rota

Veja que o link já aponta para a URL `/produtos/mostra`, que ainda não existe em nosso sistema. Vamos criá-la? Só precisamos acessar o arquivo `routes.php` e, da mesma forma como fizemos para a listagem, vincular essa URL a um método de nosso controller. O código ficará assim:

```
Route::get('/produtos/mostra', 'ProdutoController@mostra');
```

Agora, no `ProdutoController`, criamos o método `mostra`:

```
public function mostra(){
  // retorna uma view com os detalhes
}
```

Você já deve imaginar qual o próximo passo, não é? Queremos retornar uma view com os dados do produto, portanto, vamos criar um novo arquivo chamado `detalhes.php`, por enquanto em branco, dentro da pasta `/resources/views/`.

Em nosso controller, precisamos adicionar o retorno do método `mostra`, que deve renderizar essa nova view de detalhes do produto:

```
public function mostra(){
  return view('detalhes');
}
```

Até aqui nada novo, certo? Mas agora entraremos em um ponto interessante. A view, neste caso `detalhes.php`, precisa das informações do produto clicado para exibi-las em seu HTML. Algo muito parecido com o que fizemos na listagem, porém em vez de fazer um `forEach` na lista de produtos, teremos só um:

```
<h1>Detalhes do produto: <?= $p->nome ?> </h1>

<ul>
  <li>
    <b>Valor:</b> R$ <?= $p->valor ?>
  </li>
  <li>
    <b>Descrição:</b> <?= $p->descricao ?>
  </li>
  <li>
    <b>Quantidade em estoque:</b> <?= $p->quantidade ?>
  </li>
</ul>
```

Veja que utilizei um `ul` do HTML pra exibir as informações em formato de lista, mas fique à vontade para montar o HTML da forma que preferir. Utilizando a mesma estrutura da nossa view de listagem, incluindo *header* com *css*, título etc., nosso arquivo `detalhes.php` completo pode ficar assim:

```
<html>
<head>
  <link href="/css/app.css" rel="stylesheet">
  <title>Controle de estoque</title>
</head>
<body>
  <div class="container">

  <h1>Detalhes do produto: <?= $p->nome ?> </h1>
  <ul>
```

```
    <li>
      <b>Valor:</b> R$ <?= $p->valor ?>
    </li>
    <li>
      <b>Descrição:</b> <?= $p->descricao ?>
    </li>
    <li>
      <b>Quantidade em estoque:</b> <?= $p->quantidade ?>
    </li>
  </ul>

  </div>
</body>
</html>
```

Agora, em nosso controller, precisaremos deixar a variável `$p` acessível na view. Não tem erro, afinal, já fizemos isso na listagem também! Só precisamos buscar o elemento do banco de dados e depois usar o método `with` para disponibilizá-lo no `detalhes.php`. Algo como:

```
public function mostra(){
  $resposta = // busca do banco
  return view('detalhes')->with('p', $resposta);
}
```

A busca será um *select* normal, que pode ser feito com o `DB:select` que já conhecemos, mas diferente de quando queremos listar todos os elementos, agora precisaremos passar o `id` do produto como parâmetro. Ele deve ficar assim:

```
$id = 1; // precisamos pegar o id de alguma forma
$resposta =
  DB::select('select * from produtos where id = ?',[$id]);
```

Por enquanto, estamos passando o número 1 como `id` fixo, logo resolveremos isso. Veja que usamos um sinal de interrogação para mostrar o local onde o parâmetro passado no segundo argumento do método `select` (que é um array) deve ser aplicado.

Não vamos focar no SQL... agora

Não se preocupe em entender a fundo o SQL e sintaxe do `DB:select` agora, pois teremos um capítulo dedicado a isso e também veremos uma forma bem mais interessante de fazer esse tipo de consulta sem nos preocuparmos tanto em entender esses detalhes.

Outra informação importante é que o método `select` sempre retorna um `array`, que neste caso pode:

- **ter 1 elemento**, caso seja encontrado;
- **ser vazio**, caso contrário.

Precisamos portanto verificar se o `array` de resposta é vazio, mostrando uma mensagem de erro, ou caso contrário incluir o elemento da primeira posição na view. Vale lembrar que a primeira posição de um array é o índice zero, portanto o código fica assim:

```
public function mostra(){

  $id = 1; // precisamos pegar o id de alguma forma

  $resposta =
    DB::select('select * from produtos where id = ?', [$id]);

  if(empty($resposta)) {
    return "Esse produto não existe";
  }
  return view('detalhes')->with('p', $resposta[0]);
}
```

Vamos testar? Basta clicar na lupa de detalhes de qualquer produto da listagem, o resultado deverá ser parecido com:

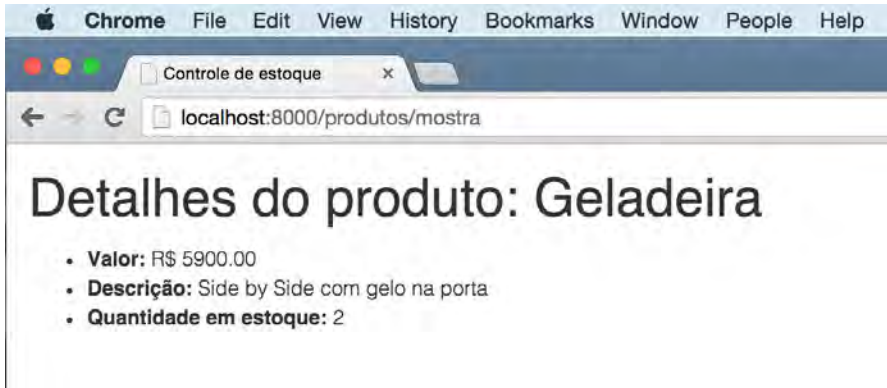



Fig. 6.3: Página de detalhes, porém sempre com o mesmo produto.

Excelente, já sabemos que nosso controller e view estão funcionando como esperado, o SQL foi executado, o elemento com `id` 1 listado e o HTML foi exibido corretamente. Mas claro, independente do produto que eu clicar, sempre estamos mostrando os dados do produto com `id` 1, que no meu caso é a *Geladeira*. Como resolver?

6.2 Enviando parâmetros na request

Como saber exatamente qual o id do produto que foi clicado no HTML? Uma forma bem tradicional de fazer isso seria passando o `id` do produto como um parâmetro na requisição, para isso bastaria mudar o link da listagem como a seguir:

```
<a href="/produtos/mostra?id=<?= $p->id ?>">
  <span class="glyphicon glyphicon-search"></span>
</a>
```

Repare que logo após a URL adicionamos um `?id=<?= $p->id ?>`, ou seja, estamos passando um parâmetro na requisição chamado `id`, com o valor do id do produto. Fazendo a alteração, você consegue verificar no código-fonte da listagem que os valores são substituídos e os links ficam assim:

```
  <h1>Listagem de produtos</h1>
▼<table class="table table-striped table-bordered table-hover">
  ▼<tbody>
    ▼<tr>
       <td> Geladeira </td>
       <td> 5900.00 </td>
       <td> Side by Side com gelo na porta </td>
       <td> 2 </td>
     ▼<td>
       ▶<a href="/produtos/mostra?id=1">…</a>      ←——  ?id=1
       </td>
     </tr>
    ▼<tr>
       <td> Fogão </td>
       <td> 950.00 </td>
       <td> Painel automático e forno elétrico </td>
       <td> 5 </td>
     ▼<td>
       ▶<a href="/produtos/mostra?id=2">…</a>      ←——  ?id=2
       </td>
     </tr>
    ▼<tr>
       <td> Microondas </td>
       <td> 1520.00 </td>
       <td> Manda SMS quando termina de esquentar </td>
       <td> 1 </td>
     ▼<td>
       ▶<a href="/produtos/mostra?id=3">…</a>      ←——  ?id=3
       </td>
     </tr>
```

Fig. 6.4: Código-fonte da listagem de produtos.

6.3 Recuperando parâmetros da request

Agora que o parâmetro `id` está sendo enviado na requisição, podemos recuperar seu respectivo valor em nosso controller com uso da interface `Request`. Por sinal, esse tipo de interface estática é conhecida como *facade*. Elas nos oferecem uma forma simples de acessar algumas das classes fundamentais do framework. Se quiser, você pode ler mais sobre esses facades em:

http://laravel.com/docs/facades

A interface `Request` tem métodos que nos auxiliam com esse trabalho, como o `input`, em que você passa a chave (*name*) do parâmetro que foi enviado na requisição e ele retorna o valor, ou *null* caso não seja encontrado.

Portanto, podemos fazer algo como:

```
$id = Request::input('id');
```

Simples, não é? Você ainda pode passar um valor default, caso o parâmetro não seja encontrado. Um exemplo seria:

```
$id = Request::input('id', '0');
```

Agora o `$id` terá o valor passado, ou zero, caso nenhum valor de `id` tenha sido enviado na requisição. Vamos fazer essa alteração? Nosso método `mostra` pode ficar assim:

```
public function mostra(){

  $id = Request::input('id', '0');

  $resposta =
    DB::select('select * from produtos where id = ?', [$id]);

  if(empty($resposta)) {
    return "Esse produto não existe";
  }
  return view('detalhes')->with('p', $resposta[0]);
}
```

Vale lembrar que, como estamos em um `namespace` diferente, temos que importar o `Request` no início de nosso controller, assim como fizemos com a interface `DB`:

```
<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;
use Request;

class ProdutoController extends Controller {
// ...
```

Tudo pronto, podemos voltar para o navegador e acessar o link de detalhes novamente que, desta vez, os detalhes do produto correto serão exibidos:

Fig. 6.5: Página de detalhes com parâmetro id.

Veja que dessa vez eu selecionei o fogão, que tem id 2, e seus dados foram exibidos conforme esperado. Também podemos testar com um `id` que não existe, mudando o final da URL pra 10, por exemplo:

Fig. 6.6: Página de detalhes com erro de produto inexistente.

Excelente, a mensagem de erro do nosso `if` foi exibida como planejado.

6.4 Conhecendo um pouco mais da Request

Vimos como é simples recuperar os parâmetros da requisição com o método `input`, passando ou não um valor default. Mas além dele, a `Request` tem uma variedade bem grande de métodos que nos ajudam em trabalhos como esse. Por exemplo, quer saber se existe um parâmetro específico na requisição? Que tal fazer:

```
if (Request::has('id'))
{
  // faz alguma coisa
}
```

Há ainda o método `all`, que retorna um array com todos os parâmetros da requisição, os métodos `only` e `except`, com que você pode restringir quais parâmetros quer listar.

```
// lista todos os params
$input = Request::all();

// apenas nome e id
$input = Request::only('nome', 'id');

// todos os params, menos o id
$input = Request::except('id');
```

Há também métodos como `url`, que retorna a URL da request atual, ou o `path` que retorna a URI. Por exemplo, em uma requisição para o método `mostra`, ao fazer:

```
$url = Request::url();
```

O valor da `$url` seria http://localhost:8000/produtos/mostra, mas já no caso do `path`:

```
$uri = Request::path();
```

O valor da `$uri` seria `produtos/mostra`.

Vamos usar mais alguns desses métodos no decorrer dos capítulos, mas se desde já quiser conhecer um pouco mais e dominar as possibilidades da interface `Request`, você pode dar uma olhada na sua documentação em:

http://laravel.com/docs/requests

6.5 URLs mais semânticas

Nossa estratégia atual de exibir o produto já resolve o problema, mas há uma alternativa bastante elegante que é adotada em situações como a nossa, onde queremos exibir detalhes de um recurso específico de nosso projeto. A ideia é que, em vez de passar o parâmetro pela *request*, dessa forma:

```
/produtos/mostra?id=2
```

você use o `id` como parte de sua rota, tendo uma URL mais semântica:

```
/produtos/mostra/2
```

Essas duas alternativas são conhecidas como parâmetro de busca (*query param*) e parâmetro de rota (*path param*), respectivamente. Normalmente a segunda estratégia é utilizada quando o parâmetro é obrigatório, como é o nosso caso, mas pode depender bastante do gosto do desenvolvedor.

Que tal aplicarmos essa alteração em nosso código? Não vai dar muito trabalho. Podemos começar pelo link, como de costume. No arquivo de listagem, em vez de fazer:

```
<a href="/produtos/mostra?id=<?= $p->id ?>">
  <span class="glyphicon glyphicon-search"></span>
</a>
```

Vamos mudar para:

```
<a href="/produtos/mostra/<?= $p->id ?>">
  <span class="glyphicon glyphicon-search"></span>
</a>
```

Veja que só tiramos o `?id=` e adicionamos uma barra no lugar. Simples, não é? Mas a alteração vai ser um pouco maior em nosso arquivo de rotas, o `routes.php`. Atualmente, a rota está assim:

```
Route::get('/produtos/mostra', 'ProdutoController@mostra');
```

Mas, na verdade, agora queremos algo como:

```
Route::get('/produtos/mostra/1', 'ProdutoController@mostra');
```

Bem, já sabe o problema, não é? O id não pode ser fixo, se não a URI sempre seria `/produtos/mostra/1`. O id precisa ser uma variável! Veja como isso é feito:

```
Route::get('/produtos/mostra/{id}', 'ProdutoController@mostra');
```

É só isso, basta envolvermos o nome do parâmetro em um par de chaves e pronto! A rota já está feita. E agora, o que falta? Vamos testar para ver se está tudo ok? Basta abrir a listagem e clicar no ícone de detalhes de qualquer produto.

Fig. 6.7: Produto não encontrado, mesmo com id correto.

Ops, o produto não existe! Isso está acontecendo com todos os produtos, o que fizemos de errado?

6.6 Recuperando parâmetros da URL

O problema está no método do controller, que atualmente usa o `Request:input` para recuperar o valor do `id`:

```
public function mostra(){
```

```
  $id = Request::input('id', '0');

  $resposta =
    DB::select('select * from produtos where id = ?', [$id]);

  if(empty($resposta)) {
    return "Esse produto não existe";
  }
  return view('detalhes')->with('p', $resposta[0]);
}
```

Como você já deve estar imaginando, o método `input` não busca parâmetros da URL, como é o caso do nosso `id` agora. No lugar disso, experimente mudar para:

```
$id = Request::route('id');
```

Agora sim, o resultado será o esperado!

Fig. 6.8: Detalhes do produto com path param.

Mas espera, e quanto ao valor default? Quando estávamos usando o método `input`, nosso código estava assim:

```
$id = Request::input('id', '0');
```

Por que não fizemos o mesmo com o `route`? A resposta é simples: o `{id}` da rota agora é obrigatório! Se não passar, não entra no método e ponto-final. Se você quiser que o id do final da url seja opcional, você precisará deixar isso explícito no momento de registrar sua rota.

```
Route::get('/produtos/mostra/{id?}', 'ProdutoController@mostra');
```

Repare que há uma `?` após o id indicando que ele é opcional. Mas, claro, em nosso caso não queremos isso. O id sempre precisará ser passado.

Isso não é tudo, o Laravel nos oferece uma forma ainda mais legal de recuperar parâmetros da URL. Quer ver como? Basta adicionar um argumento **com o mesmo nome do parâmetro** na assinatura do seu método, ele vai ser populado como um passe de mágica. Veja como fica o método do controller:

```
public function mostra($id){

    $resposta =
    DB::select('select * from produtos where id = ?', [$id]);

    if(empty($resposta)) {
        return "Esse produto não existe";
    }
    return view('detalhes')->with('p', $resposta[0]);
}
```

Note que em nenhum momento precisamos usar a `Request`, o framework faz esse trabalho por nós. Abra a página novamente para testar, o resultado será o mesmo:

Fig. 6.9: Página de detalhes do produto.

6.7 Alguns cuidados necessários

Quando estamos usando parâmetros na URL, sempre precisamos nos atentar a alguns detalhes. Por exemplo, em algum momento falamos que o `id` precisaria ser um número? Não. Isso significa que eu consigo acessar o método `mostra` pela seguinte URL:

```
http://localhost:8000/produtos/mostra/teste
```

Claramente isso não deveria acontecer. Por nossa sorte, a única consequência aqui será ver a mensagem de que o produto não foi encontrado. Mas existem problemas piores, como ambiguidade entre rotas.

Em nosso caso, isso seria um pouco difícil, mas imagine que para simplificar optássemos por remover o `/mostra` de nossa URL. Isso causaria sérios problemas, pois as seguintes URLs seriam equivalentes:

- http://localhost:8000/produtos/1
- http://localhost:8000/produtos/adiciona

Já que o `{id}` pode ser qualquer coisa, inclusive um texto, ele pode ser a palavra `adiciona`. Em outras palavras, dependendo da ordem em que eu registrar as rotas, pode acontecer de eu acessar `/produtos/adiciona` e a aplicação me responder que esse produto não existe... Ops!

Nesse caso, precisaríamos de alguma forma ensinar ao Laravel que o {id} da rota sempre será um número. Isso pode ser feito com auxilio do método where, como no exemplo:

```
Route::get(
  '/produtos/mostra/{id}',
  'ProdutoController@mostra'
  )
  ->where('id', '[0-9]+');
```

Observe que estamos passando para o método where o nome do parâmetro e uma expressão regular (*regex*) com o *pattern* que pode ser seguido. Faça a alteração e tente acessar a URL http://localhost:8000/produtos/mostra/teste novamente. O resultado dessa vez será uma NotFoundHttpException.

Fig. 6.10: NotFoundHttpException na URL com param não numérico.

Justo, não é? Afinal, essa URL realmente não existe. Repare que, passando um número válido, tudo continua funcionando.

Capítulo 7

Views mais flexíveis e poderosas

Um dos problemas de nosso código atual é que existe muita repetição em nossas views. Veja, por exemplo, como está nosso arquivo de `listagem.php`:

```
<html>
<head>
  <link href="/css/app.css" rel="stylesheet">
  <title>Controle de estoque</title>
</head>
<body>
  <div class="container">

    <h1>Listagem de produtos</h1>
    <table class="table table-striped ...">
    <!-- conteúdo omitido -->
```

```
  </div>
</body>
</html>
```

Agora perceba a semelhança com o HTML do arquivo `detalhes.php`:

```
<html>
<head>
  <link href="/css/app.css" rel="stylesheet">
  <title>Controle de estoque</title>
</head>
<body>
  <div class="container">

  <h1>Detalhes do produto: <?= $p->nome ?> </h1>
  <!-- conteúdo omitido -->

  </div>
</body>
</html>
```

O conteúdo da *div* principal é diferente, afinal, cada página tem suas regras e objetivos específicos, mas todo o resto está igual! Com isso, no dia a dia se queremos criar uma nova página, sem pensar duas vezes acabamos copiando e colando alguma que existe, mudando apenas o miolo. Quem nunca fez isso?

O problema dessa abordagem é que, hora ou outra, você pode querer mudar o cabeçalho, menu, ou qualquer outro elemento que foi copiado e colado em quase todas as suas páginas. E agora? Replico a mudança nas minhas 200 views manualmente? Não parece uma solução ideal, não é?

Neste capítulo veremos como o Laravel pode nos ajudar nesse trabalho, tornando nossas views mais flexíveis, evitando repetições e ainda ganhando poderosos recursos que nos ajudam nas diferentes necessidades do nosso cotidiano.

7.1 Blade, a template engine do Laravel

Se o problema estivesse acontecendo no código de nosso controller ou em nossas regras de negócio, bastaria extrair o código repetido pra um arquivo ou classe e pronto, poderíamos reutilizá-la sempre. Que tal se fosse possível fazer o mesmo na view, extraindo toda essa estrutura principal em um único arquivo e reutilizá-lo sempre que quiséssemos? Pois bem, isso não só é possível como também **muito recomendado**. Dessa forma, quando quisermos adicionar algum link, mudar os elementos ou fazer qualquer alteração nessa parte em comum, mudamos em um único lugar, e todo o resto é atualizado. O ganho em manutenibilidade é enorme.

Quem nos ajuda com esse trabalho é o **Blade**, a *template engine* do Laravel. Quer ver como é simples? Para resolver esse problema de repetição, basta criar um novo arquivo com o código em comum, que será nosso *layout* principal.

Um fator importante é que todo arquivo do blade precisar ter a extensão `.blade.php`, portanto, vamos chamá-lo de `principal.blade.php`, e lembre-se de que, como é uma *view*, precisa estar no diretório `/resources/views`.

Fig. 7.1: Layout do blade dentro da pasta /resources/views.

Dentro desse arquivo, vamos colocar todo o HTML em comum, e utilizar a marcação `@yield` com o nome da seção que deverá ser sobrescrita pelas outras views. A ideia é simples, suas views "herdam" desse layout principal, e sobrescrevem as seções nele definidas:

Fig. 7.2: Layout principal com seção de conteúdo.

O código do arquivo `principal.blade.php` pode ficar assim:

```
<html>
<head>
    <link href="/css/app.css" rel="stylesheet">
    <title>Controle de estoque</title>
</head>
<body>
  <div class="container">

    @yield('conteudo')

  </div>
</body>
</html>
```

Nada muito diferente, não é? Tem todo o HTML em comum, e a marcação `@yield(‘conteudo’)` para delimitar onde o conteúdo das outras views deve ser inserido. Vamos testar? O primeiro passo será mudar a extensão do arquivo `detalhes.php` para `detalhes.blade.php`.

Fig. 7.3: Arquivo de detalhes com extensão do blade.

Feito isso, podemos tirar todo o código que já existe no layout `principal.blade.php`, e deixar apenas o miolo que estava dentro da *div* com classe *container*. O arquivo vai ficar apenas com este conteúdo:

```
<h1>Detalhes do produto: <?= $p->nome ?> </h1>

<ul>
  <li>
    <b>Valor:</b> R$ <?= $p->valor ?>
  </li>
  <li>
    <b>Descrição:</b> <?= $p->descricao ?>
  </li>
  <li>
    <b>Quantidade em estoque:</b> <?= $p->quantidade ?>
  </li>
</ul>
```

Ótimo! Mas ainda precisamos dizer que esse arquivo deve utilizar o layout principal, do arquivo `principal.blade.php`. Para isso, basta adicionar a marcação `@extends` com o nome do layout que ele deve herdar (neste caso: `principal`, que é o nome do arquivo sem a extensão).

```
@extends('principal')

<!-- restante do html aqui -->
```

Também é preciso colocar todo o conteúdo restante do nosso arquivo de detalhes dentro de uma `@section`, com o nome que foi definido no `@yield` do layout principal, que neste caso é `conteudo`:

```
@extends('principal')

@section('conteudo')
<h1>Detalhes do produto: <?= $p->nome ?> </h1>

<ul>
  <li>
    <b>Valor:</b> R$ <?= $p->valor ?>
  </li>
  <li>
    <b>Descrição:</b> <?= $p->descricao ?>
  </li>
  <li>
    <b>Quantidade em estoque:</b> <?= $p->quantidade ?>
  </li>
</ul>

@stop
```

Veja que usamos um `@stop` para marcar o fim da `section`. Isso é necessário pois, em alguns momentos, podemos querer sobrescrever mais de uma `section` do layout principal.

Fig. 7.4: Visão geral da hierarquia de views.

Vamos ver se após a mudança tudo continua funcionando? Basta acessar a página de detalhes de qualquer produto, por exemplo:

http://localhost:8000/produtos/mostra/1

Fig. 7.5: Página de detalhes agora com blade.

Veja que, como esperado, a página continua funcionando normalmente. Se quiser, clique com o botão direito na página e selecione a opção *view page source*. Repare que o HTML continua o mesmo!

Fig. 7.6: Código HTML da página de detalhes renderizado após a migração para blade.

Legal, não é? Mas o **blade** nos oferece bem mais que isso! Temos muito mais o que explorar; vamos ver as formas de exibir o valor de nossas variáveis.

7.2 Variáveis com blade

Atualmente estamos usando a marcação do PHP puro para imprimir os valores do produto. Quando fazemos:

```
<li>
  <b>Valor:</b> R$ <?= $p->valor ?>
</li>
```

Não há problema nisso e, se quiser, você ainda pode continuar exibindo dessa forma. Mas se quiser e eu recomendo , você também pode fazer da forma do blade, envolvendo a variável em chaves duplas como a seguir:

```
<li>
  <b>Valor:</b> R$ {{ $p->valor }}>
</li>
```

A mudança é simples e o resultado será o mesmo, mas a sintaxe, em minha opinião, é um pouco mais agradável. Entretanto, o ganho vai além de uma sintaxe melhor, ou um caractere a menos. Você ganha bastante flexibilidade e alguns recursos extras, como o `or`, para definir um valor *default* caso a variável não esteja populada. Um exemplo seria:

```
{{$p->descricao or 'nenhuma descrição informada'}}
```

Legal, não é? Vamos conhecer diversos outros recursos no decorrer deste capítulo e livro, mas por enquanto, já podemos fazer a mudança no restante do arquivo, que deve ficar assim:

```
@extends('principal')

@section('conteudo')

<h1>Detalhes do produto: {{$p->nome}} </h1>

<ul>
  <li>
    <b>Valor:</b> R$ {{$p->valor}}
  </li>
  <li>
    <b>Descrição:</b> {{$p->descricao}}
  </li>
  <li>
    <b>Quantidade em estoque:</b> {{$p->quantidade}}
  </li>
</ul>

@stop
```

7.3 Migrando a listagem para Blade

Vamos fazer a mesma alteração no arquivo `listagem.php`? Os passos serão os mesmos, começando por renomear o arquivo, que agora deve se chamar `listagem.blade.php`. Além disso, já podemos tirar todo o código que já está definido no layout principal e adicionar o `@extends` e `@section`. A listagem deve ficar assim:

```
@extends('principal')

@section('conteudo')

  <h1>Listagem de produtos</h1>
  <table class="table table-striped table-bordered table-hover">
    <?php foreach ($produtos as $p): ?>
    <tr>
      <td><?= $p->nome ?></td>
      <td><?= $p->valor ?></td>
      <td><?= $p->descricao ?></td>
      <td><?= $p->quantidade ?></td>
      <td>
        <a href="/produtos/mostra/<?= $p->id ?>">
          <span class="glyphicon glyphicon-search"></span>
        </a>
      </td>
    </tr>
    <?php endforeach ?>
  </table>

@stop
```

Legal, não é? O arquivo já fica um pouco mais limpo. Já podemos também mudar a forma como estamos exibindo nossas variáveis, utilizando as chaves duplas do blade. O corpo da tabela ficará assim:

```
<tr>
  <td>{{ $p->nome }} </td>
  <td>{{ $p->valor }} </td>
  <td>{{ $p->descricao }} </td>
```

```
  <td>{{ $p->quantidade }} </td>
  <td>
    <a href="/produtos/mostra/{{$p->id}}>">
      <span class="glyphicon glyphicon-search"></span>
    </a>
  </td>
</tr>
```

7.4 Loopings e mais loopings

O blade também nos oferece uma maneira bastante interessante de enxugar a sintaxe do nosso `foreach`, na listagem de produtos. Veja que atualmente estamos fazendo com PHP puro, ou seja, utilizando a tag `<?php ?>`. Isso funciona, e você pode continuar fazendo assim se preferir, claro, mas que tal tirar toda essa verbosidade e adicionar um simples `@` antes desse operador? Em outras palavras, em vez de fazer:

```
<?php foreach ($produtos as $p): ?>
  <!-- codigo omitido -->
<?php endforeach ?>
```

Você pode mudar para:

```
@foreach ($produtos as $p)
  <!-- codigo omitido -->
@endforeach
```

Isso melhora bastante a legibilidade do código. Agora o arquivo `listagem.blade.php` completo deve estar parecido com:

```
@extends('principal')

@section('conteudo')

  <h1>Listagem de produtos</h1>
  <table class="table table-striped table-bordered table-hover">
    @foreach ($produtos as $p)
    <tr>
      <td> {{$p->nome}} </td>
```

```
            <td> {{$p->valor}} </td>
            <td> {{$p->descricao}} </td>
            <td> {{$p->quantidade}} </td>
            <td>
              <a href="/produtos/mostra/{{$p->id}}>">
                <span class="glyphicon glyphicon-search"></span>
              </a>
            </td>
          </tr>
          @endforeach
        </table>

@stop
```

Outros tipos de looping

Além do `foreach`, também podemos fazer o mesmo com o `for` tradicional:

```
@for ($i = 0; $i < 10; $i++)
    O indice atual é {{ $i }}
@endfor
```

Ou ainda com um `while`:

```
@while (true)
    Entrando em looping infinito!
@endwhile
```

Ha ainda uma variação bastante interessante, chamada `forelse`. Se a lista for vazia, ele executa o código do bloco marcado com `@empty`:

```
@forelse($produtos as $p)
    <li>{{ $p->nome }}</li>
@empty
    <p>Não tem nenhum produto!</p>
@endforelse
```

Legal, não é? Mas como estamos fazendo o `foreach` no miolo de uma tabela, não seria muito interessante. No lugar disso, podemos usar um `if`!

7.5 Adicionando condições na view

Podemos fazer um `if` verificando se a lista de produtos está vazia, e se estiver, mostrar uma mensagem de que não existe nenhum produto cadastrado. Em PHP puro, o código ficaria assim:

```
<?php if(empty($produtos)) { ?>

<div class="alert alert-danger">
  Você não tem nenhum produto cadastrado.
</div>

<?php } else { ?>

  <h1>Listagem de produtos</h1>
  <table>

  <? foreach ($produtos as $p) { ?>
?>
<!-- continuação do código -->

<?php } ?> <!-- fechando o foreach -->
<?php } ?> <!-- fechando o else -->
```

Um pouco confuso e difícil de ler. Que tal com blade?

```
@if(empty($produtos))

<div class="alert alert-danger">
  Você não tem nenhum produto cadastrado.
</div>

@else

  <h1>Listagem de produtos</h1>
  <table>

  @foreach ($produtos as $p)
  <!-- continuação do código -->
  @endforeach
```

```
@endif
```

A ausência das chaves e tags do PHP ajuda bastante, nosso código fica mais simples e legível. Vamos fazer a alteração? O arquivo `listage.blade.php` completo pode ficar assim:

```
@extends('principal')

@section('conteudo')

 @if(empty($produtos))
  <div>Você não tem nenhum produto cadastrado.</div>

 @else
  <h1>Listagem de produtos</h1>
  <table class="table table-striped table-bordered table-hover">
    @foreach ($produtos as $p)
    <tr>
      <td> {{$p->nome}} </td>
      <td> {{$p->valor}} </td>
      <td> {{$p->descricao}} </td>
      <td> {{$p->quantidade}} </td>
      <td>
        <a href="/produtos/mostra/{{$p->id}}>">
          <span class="glyphicon glyphicon-search"></span>
        </a>
      </td>
    </tr>
    @endforeach
  </table>

  @endif
@stop
```

Vamos testar? Para não ter que apagar os produtos do banco, vamos roubar um pouco, incluindo um array vazio no método `lista` do nosso controller:

```
class ProdutoController extends Controller {

  public function lista(){
    // busca os produtos do banco
    return view('listagem')->with('produtos', array());
  }

// continuação do código omitido
```

Com isso, ao acessar a listagem, o resultado será:

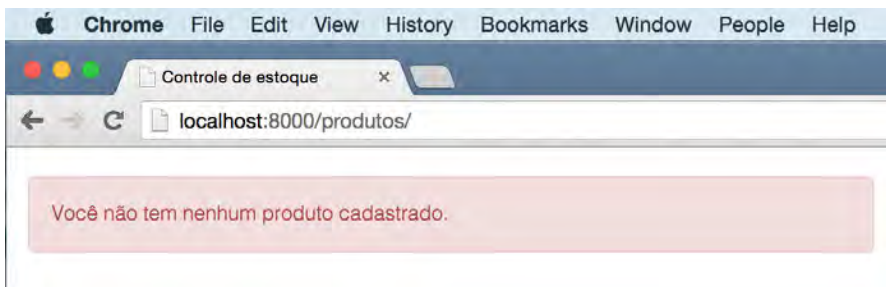



Fig. 7.7: Listagem de produtos com mensagem de lista vazia.

Excelente, tudo está funcionando como esperamos! Agora que testamos, não se esqueça de voltar o método `lista` ao normal, passando o array de `$produtos` no lugar de um array vazio.

Outros formas de condicionar

Além do `@if` e `@else`, você também pode usar `@elseifs` ou mesmo o `@unless`, que faz a condição inversa. Por exemplo, no caso a seguir, o valor condicionado sempre será exibido:

```
@unless (1 == 2)
  Esse texto sempre será exibido!
@endunless
```

7.6 Marcando produtos em falta no estoque

Queremos agora pintar de vermelho as colunas da tabela que contêm produtos com quantidade menor ou igual a 1. Essa marcação visual vai nos ajudar a perceber quando um produto está em falta em nosso estoque.

Pra colorir a linha, podemos adicionar a classe `danger`, do bootstrap, nos `trs` dos produtos em falta. Apenas para ver o efeito da alteração, vamos adicionar a classe em todas as colunas:

```
<table class="table table-striped ...">
    @foreach ($produtos as $p)
    <tr class="danger">
        <!-- código omitido -->
    </tr>
    @endforeach
```

Atualize a página da listagem para ver a diferença, ela agora deve estar parecida com:

Fig. 7.8: Listagem com todos os produtos marcados de vermelho.

Ótimo, agora só falta condicionar quais os produtos que devem ficar pintados de vermelho. Claro, um `@if` resolveria, mas ficaria bem feio e verboso. Em vez disso, podemos fazer uma condição parecida com:

```
{{ $p->quantidade <= 1 ? 'danger' : ''}}
```

Esse é o tão conhecido ternário que, como você já deve ter percebido, tem a seguinte estrutura:

```
{{ condicao ? 'valor_se_true' : 'valor_se_false'}}
```

Ou seja, se a condição booleana for verdadeira (*true*), o primeiro valor será devolvido. Caso contrário, retornará o segundo valor. É importante saber que esse não é um recurso exclusivo do blade, claro, em PHP puro você também pode fazer:

```
<?php $p->quantidade <= 1 ? 'danger' : '' ?>
```

O que mudou foi a tag, assim como nos demais exemplos, mas nesse caso, usando o blade não precisamos adicionar nenhum `@` para que isso funcione, basta colocar o ternário dentro das chaves duplas e pronto. Vamos testar? Basta adicionar essa condição dentro da `class` do `tr`. A tabela vai ficar assim:

```
<table class="table table-striped ...">
  @foreach ($produtos as $p)
  <tr class="{{$p->quantidade<=1 ? 'danger' : '' }}">
    <!-- código omitido -->
  </tr>
  @endforeach
</table>
```

Pronto para testar? É só acessar a listagem novamente! No meu caso, ela ficará assim:

Fig. 7.9: Listagem de produtos com verificação de quantidade em estoque.

Também podemos adicionar uma legenda para deixar essa informação mais intuitiva. Vamos colocar um *span* logo depois do `@endif`, no final da tabela:

```
<h4>
  <span class="label label-danger pull-right">
    Um ou menos itens no estoque
  </span>
</h4>
```

Veja que ele já tem algumas classes do bootstrap para estilizá-lo. Com essa alteração, o arquivo `listagem.blade.php` completo deve ficar assim:

```
@extends('principal')

@section('conteudo')

 @if(empty($produtos))
  <div class="alert alert-danger">
    Você não tem nenhum produto cadastrado.
  </div>

 @else
```

```
  <h1>Listagem de produtos</h1>
  <table class="table table-striped table-bordered table-hover">
    @foreach ($produtos as $p)
    <tr class="{{$p->quantidade<=1 ? 'danger' : '' }}">
      <td> {{$p->nome}} </td>
      <td> {{$p->valor}} </td>
      <td> {{$p->descricao}} </td>
      <td> {{$p->quantidade}} </td>
      <td>
        <a href="/produtos/mostra/{{$p->id}}>">
          <span class="glyphicon glyphicon-search"></span>
        </a>
      </td>
    </tr>
    @endforeach
  </table>
 @endif

 <h4>
  <span class="label label-danger pull-right">
    Um ou menos itens no estoque
  </span>
 </h4>

@stop
```

Acesse a listagem novamente para ver o resultado! Ela deve estar parecida com:

Fig. 7.10: Aparência final da listagem com legenda.

7.7 Organizando nossas views

Conforme nosso sistema vai crescendo, mais e mais views devem aparecer e precisamos organizá-las de alguma forma. É muito comum separar esses arquivos por controller, ou qualquer estrutura que facilite seu trabalho e também o trabalho de sua equipe.

Um exemplo de separação por controller seria, se temos um `ProdutoController`, criar um diretório chamado `produto` e colocar nele todas as **views** relacionadas. O mesmo para usuários e qualquer outro elemento que venha a aparecer em nossa aplicação.

Vamos criar uma nova pasta chamada `produto`, dentro de `resources/views`, e mover para ela os arquivos `listagem.blade.php` e `detalhes.blade.php`. Seu projeto deve ficar assim:

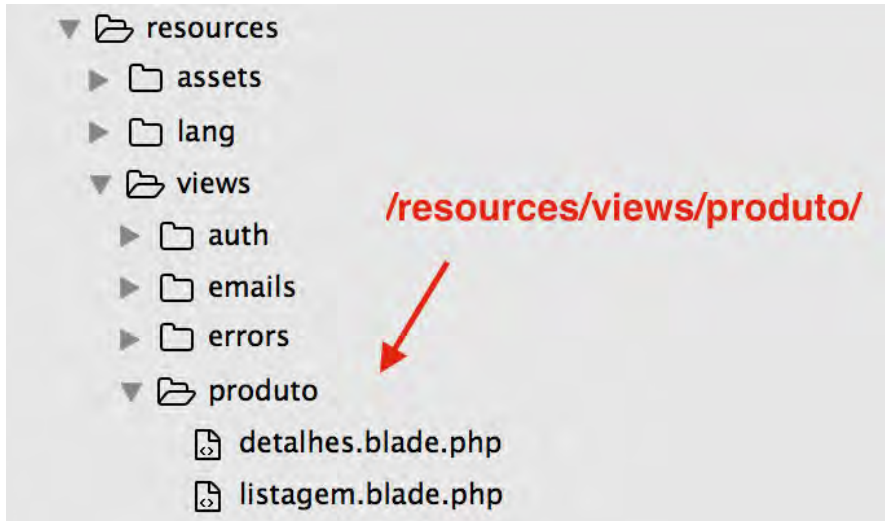



Fig. 7.11: Pasta produto com arquivos de listagem e detalhes.

Além disso, vamos criar uma nova pasta chamada `layout`, também em `resources/views`, para mover o arquivo `principal.blade.php`:

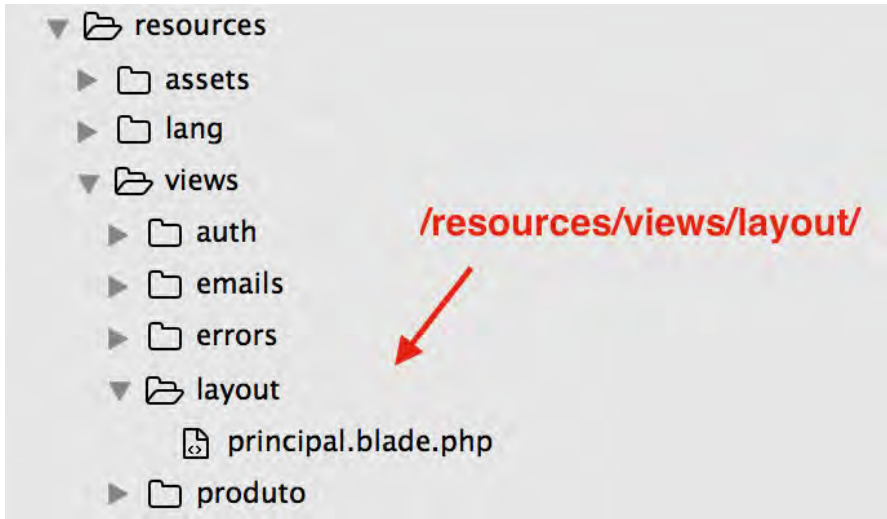



Fig. 7.12: Pasta layout com arquivo principal.

Mas, agora que mudamos isso, tente acessar qualquer página da sua aplicação. O resultado, talvez já esperado, será um erro!

Fig. 7.13: Exception de arquivo não encontrado.

A mensagem já dá uma pista: **View listagem not found**. Agora que mudamos a view para um subdiretório, precisamos alterar o retorno dos métodos

do nosso controller. Atualmente fazemos algo como:

```
return view('listagem');
```

Em vez disso, precisamos ensinar ao Laravel que a view está dentro da pasta `produto`:

```
return view('produto/listagem');
```

Isso já funciona, mas é ainda mais comum fazermos desta forma:

```
return view('produto.listagem');
```

Veja que usamos um ponto no lugar da barra. O resultado será o mesmo, você pode escolher o que preferir. Nosso controller deve ficar assim:

```
class ProdutoController extends Controller {

  public function lista(){
    // código omitido
    return view('produto.listagem')->with('produtos', $produtos);
  }

  public function mostra($id){
    // código omitido
    return view('produto.detalhes')->with('p', $produto);
  }
```

Vamos testar? Acessamos a página novamente e... ops!

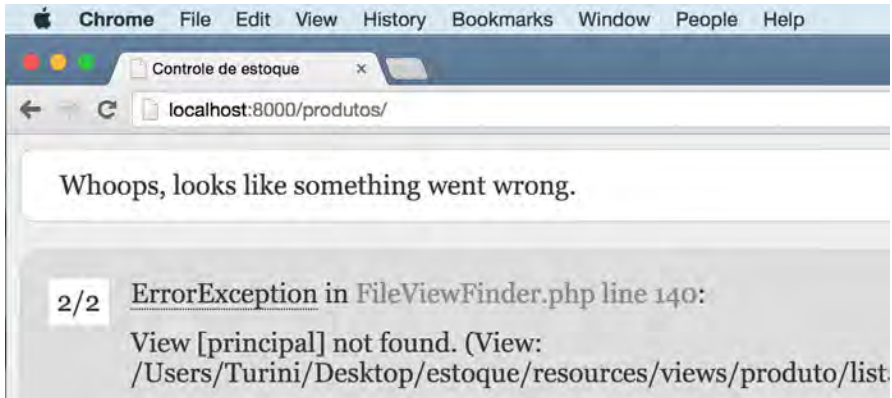



Fig. 7.14: Exception de arquivo não encontrado.

O arquivo `principal.blade.php` não foi encontrado nas views, também precisamos mudar o lugar onde usamos o `@extends`! Agora, no lugar de `@extends('principal')`, precisamos mudar para `@extends('layout.principal')`. A alteração deve ser feita nos arquivos `detalhes.blade.php` e `listagem.blade.php`:

```
@extends('layout.principal')

@section('conteudo')

<!-- restante do conteúdo omitido -->
```

Ótimo, com isso nossa aplicação volta a funcionar como esperado.

7.8 Estilizando o template principal

Como nossas views seguem um template padrão, definido no `principal.blade.php`, agora fica muito mais fácil evoluir e estilizar o design da nossa aplicação web. Podemos adicionar um header com o nome do projeto, links, footer com mensagem de direitos autorais etc.

Veja como é simples. Como o layout está isolado, só precisamos aplicar as alterações no `principal.blade.php` e ela será propagada em todas as páginas. Um exemplo de layout seria:

```
<html>
<head>
    <link href="/css/app.css" rel="stylesheet">
    <link href="/css/custom.css" rel="stylesheet">
    <title>Controle de estoque</title>
</head>
<body>
  <div class="container">

  <nav class="navbar navbar-default">
    <div class="container-fluid">

    <div class="navbar-header">
      <a class="navbar-brand" href="/produtos">
        Estoque Laravel
      </a>
    </div>

      <ul class="nav navbar-nav navbar-right">
        <li><a href="/produtos">Listagem</a></li>
      </ul>

    </div>
  </nav>

    @yield('conteudo')

  <footer class="footer">
      <p>© Livro de Laravel da Casa do Código.</p>
  </footer>

  </div>
</body>
</html>
```

Repare que estamos importando um arquivo `custom.css`, que ainda não existe. Se quiser, você pode criar o arquivo dentro de `/public/css/`, com o conteúdo a seguir:

```
nav.navbar.navbar-default{
  background-color: #337ab7;
  border-color: #2e6da4;
  margin-top: 10px;
}

nav.navbar.navbar-default a {
  color: white;
}

li p.navbar-text, a.navbar-brand {
  color: white!important;
  font-weight: bolder;
}

footer.footer {
  background-color: #337ab7;
  border-color: #2e6da4;
  margin-top: 55px;
  color: white;
  border-radius: 5px;
}

footer.footer p {
  padding: 10px;
}
```

Com esse estilo, a listagem vai ficar parecida com:

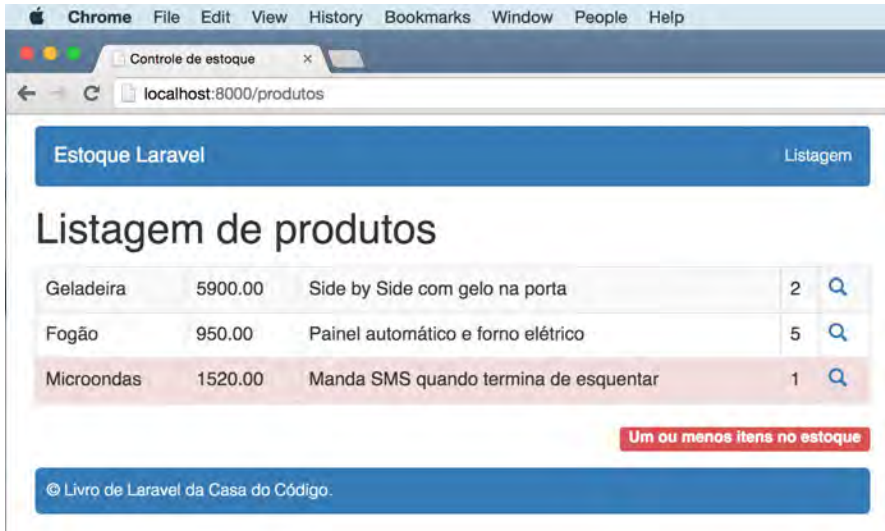



Fig. 7.15: Aparência final da listagem.

E a view de detalhes deve ficar assim:

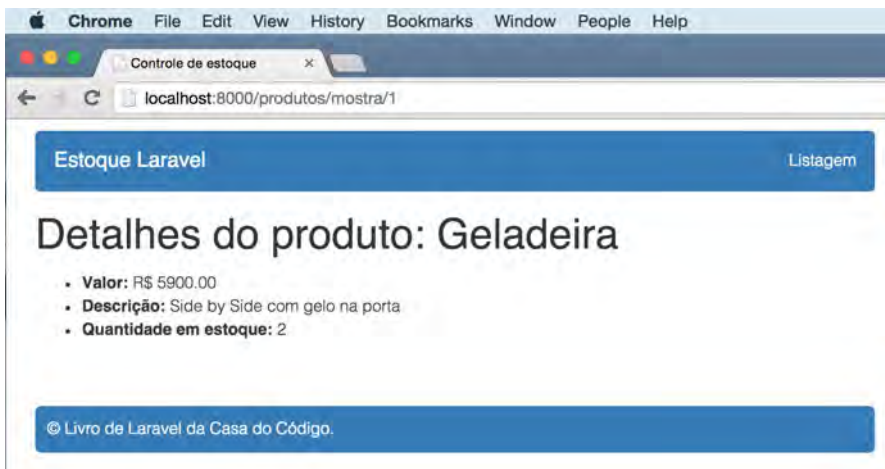



Fig. 7.16: Aparência final da view de detalhes.

Legal, não é? Mudamos um único arquivo e o estilo foi propagado em

todos. Claro, fique à vontade para estilizar o HTML da forma que preferir, você pode e deve ir sempre além do que vemos aqui no livro.

Capítulo 8

Request e métodos HTTP

8.1 Criando formulário de novos produtos

Já somos capazes de visualizar uma lista com todos os produtos e os detalhes de um produto específico, mas ainda não temos uma função de adicionar novos produtos em estoque. Vamos implementá-la?

Podemos começar pela rota, no arquivo `routes.php`. Vamos registrar uma URI `/produtos/novo` apontando para o método `novo`, do `ProdutoController`. Ela pode ficar assim:

```
Route::get('/produtos/novo', 'ProdutoController@novo');
```

Vale lembrar que, antes de adicionar um produto no banco de dados, precisamos abrir uma página com formulário para que o usuário possa preencher os dados, portanto é isso o que o método `novo` fará. Ele retornará uma view chamada `formulario`, que será criada dentro da pasta `produto`:

```
public function novo(){
    return view('produto.formulario');
}
```

Para fechar esse passo inicial, precisamos criar o arquivo `formulario.blade.php` dentro de `/resources/view/produtos/`. Como queremos que ele use o layout padrão, no início do arquivo declaramos que ele herda do layout principal e declaramos a `section` de conteúdo, assim como fizemos com os demais arquivos. Já vamos adicionar um `form` também. O arquivo deve ficar assim:

```
@extends('templates.principal')

@section('conteudo')

<h1>Novo produto</h1>

<form>
  <label>Nome</label>
  <input/>
  <label>Descricao</label>
  <input/>
  <label>Valor</label>
  <input/>
  <label>Quantidade</label>
  <input type="number"/>
  <button type="submit">Submit</button>
</form>

@stop
```

Vamos testar para garantir que tudo funcionou até aqui? Só precisamos acessar a URL http://localhost:8000/produtos/novo. O resultado será:

Fig. 8.1: Formulário sem estilos de CSS.

O formulário está um pouco estranho, com as *labels* e *inputs* todos bagunçados em uma mesma linha. Já que estamos usando bootstrap, podemos usar um de seus modelos de formulários que pode ser encontrado em sua documentação:

http://getbootstrap.com/css/#forms

Fazendo a alteração, o HTML poderá ficar assim:

```
@extends('templates.principal')

@section('conteudo')

<h1>Novo produto</h1>

<form>
  <div class="form-group">
    <label>Nome</label>
    <input class="form-control">
  </div>
  <div class="form-group">
    <label>Descricao</label>
    <input class="form-control">
```

```
  </div>
  <div class="form-group">
    <label>Valor</label>
    <input class="form-control">
  </div>
  <div class="form-group">
    <label>Quantidade</label>
    <input type="number" class="form-control">
  </div>
  <button type="submit" class="btn
    btn-primary btn-block">Submit</button>
</form>

@stop
```

Vamos ver o resultado?

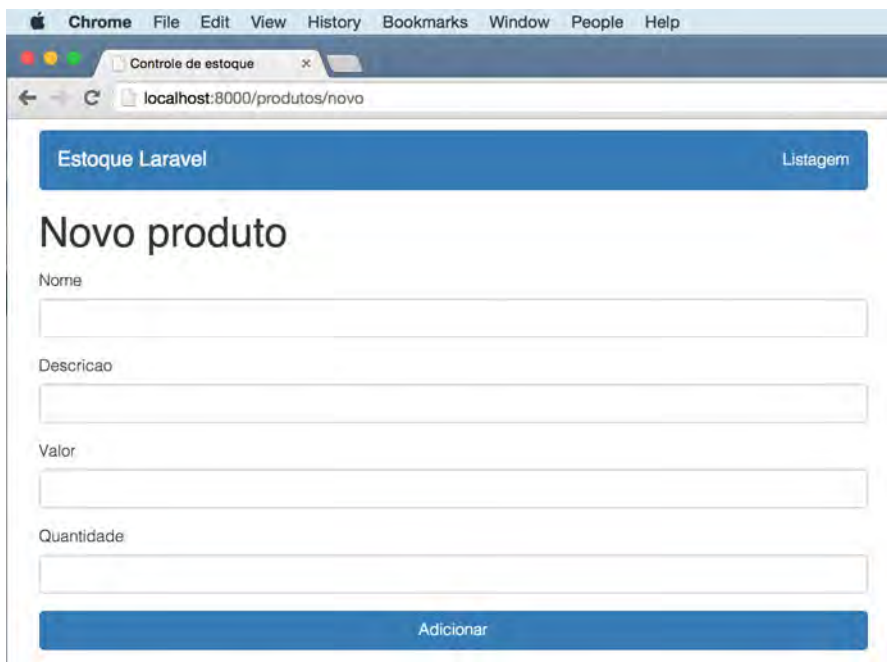



Fig. 8.2: Formulários com estilos CSS do bootstrap.

Excelente! Agora que temos uma nova página, podemos adicionar um novo link no cabeçalho do arquivo `principal.blade.php`. O HTML com o novo link ficará assim:

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="/produtos">Listagem</a></li>
  <li><a href="/produtos/novo">Novo</a></li>
</ul>
```

Com isso boa parte do trabalho está pronto, mas agora precisamos implementar a função que efetivamente adicionará um novo produto ao banco.

8.2 Criando o método adiciona

Como de costume, podemos começar pelo mapeamento desse método, que deve ser feito no `routes.php`. Podemos registrar uma nova rota `/produtos/adiciona`, que apontará para o método `adiciona` do `ProdutoController`. O código ficará assim:

```
Route::get('/produtos/adiciona', 'ProdutoController@adiciona');
```

Agora que já temos a rota, precisamos criar esse novo método no controller. Sua assinatura ficará desta forma:

```
public function adiciona(){
  // deve adicionar os produtos no banco
}
```

E agora, qual o próximo passo? Antes mesmo de pensar no SQL que deve ser executado no banco de dados, ou na view que deve ser retornada, precisamos pegar os parâmetros digitados no formulário de alguma maneira! Os passos necessários para o método `adiciona` serão:

```
public function adiciona(){
  // pegar dados do formulario
  // salvar no banco de dados
  // retornar alguma view
}
```

Já vimos que é a classe `Request` quem cuida desse trabalho, de pegar os parâmetros da requisição. Portanto podemos fazer algo como:

```
$nome = Request::input('o_que_passar_aqui_?');
```

Eis a questão. Quando estávamos buscando o parâmetro enviado pelo link de visualizar, sabíamos que a chave era `id`, pois ela era enviada pela URL dessa forma:

```
/produtos/mostra?id=1
```

Veja que estamos deixando explícito que o parâmetro se chama `id`, e que ele recebe o valor 1. Como agora temos um formulário, e não um link, precisaremos adicionar um `name` em cada *input* do formulário para que seja possível recuperar suas informações pelo controller. O HTML completo ficará assim:

```
@extends('templates.principal')

@section('conteudo')

<h1>Novo produto</h1>

<form action="/produtos/adiciona">
  <div class="form-group">
    <label>Nome</label>
    <input name="nome" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Descricao</label>
    <input name="descricao" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Valor</label>
    <input name="valor" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Quantidade</label>
    <input type="number" name="quantidade" class="form-control"/>
  </div>
```

```
  <button type="submit"
    class="btn btn-primary btn-block">Submit</button>
</form>

@stop
```

Com isso, em nosso controller, podemos recuperar cada um dos valores:

```
public function adiciona(){

  $nome = Request::input('nome');
  $descricao = Request::input('descricao');
  $valor = Request::input('valor');
  $quantidade = Request::input('quantidade');

  // salvar no banco de dados
  // retornar alguma view
}
```

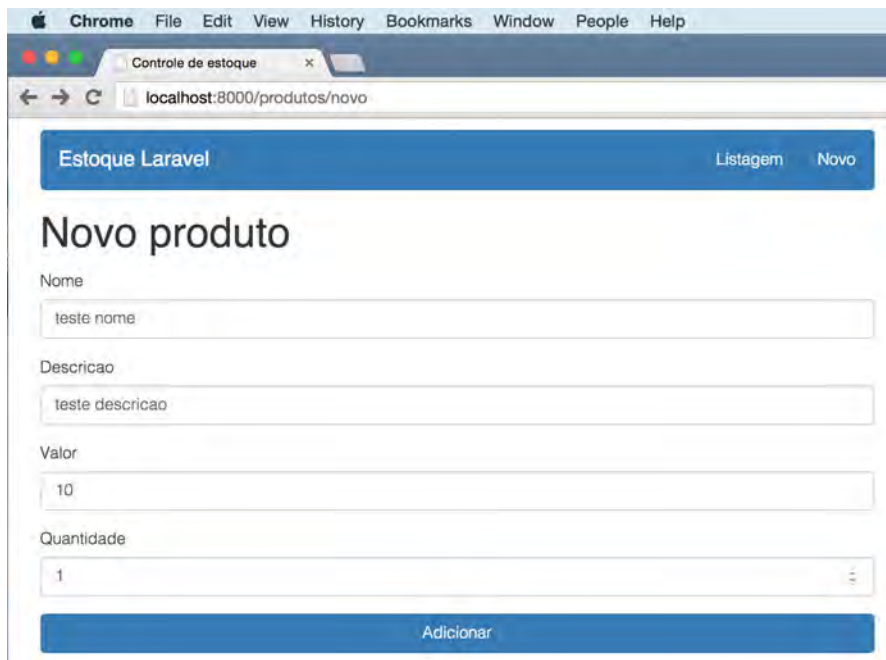
Outras alternativas

Podemos sim utilizar o método `all` da `Request` para pegar todos os valores de uma única vez em um `array`, ou ainda o método `only` deixando explícito quais parâmetros queremos buscar:

```
$all = Request::all();
$only = Request::only('nome', 'valor', '...');
```

Mas para deixar bem explícito, e garantir a ordem dos dados, por enquanto faremos dessa forma. Voltaremos a falar sobre esse método para melhorar este e outros pontos que serão detalhados mais à frente.

Ótimo, com isso já temos cada um dos valores digitados no formulário. Vamos testar? Podemos mudar o método `adiciona` para retornar esses valores:

```
public function adiciona(){
```

```
  $nome = Request::input('nome');
  $descricao = Request::input('descricao');
  $valor = Request::input('valor');
  $quantidade = Request::input('quantidade');

  return implode( ', ', array($nome,
    $descricao, $valor, $quantidade));
}
```

O `implode` fará com que cada um dos valores seja impresso separado por vírgula. Agora só falta um detalhe, precisamos adicionar a `action` do formulário, para que ele envie para o método `adiciona`:

```
<form action="/produtos/adiciona">
<!-- restante do html omitido -->
```

Vamos testar? Basta abrir o formulário no navegador e preencher os campos com algum valor de teste:

Fig. 8.3: Formulário preenchido com valores de teste.

Ao clicar no botão *Adicionar*, a saída será:

Fig. 8.4: Mensagem com informações do produto adicionado.

8.3 Inserindo os produtos no BD

Boa parte do trabalho está pronta, mas ainda precisamos persistir o produto no banco de dados. Por enquanto, faremos isso com a função `insert` da interface `DB`, que já usamos no método `lista`. O código de adicionar ficará parecido com:

```
DB::insert('insert into produtos
  (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));
```

Veja que, assim como fizemos no *select* do método `mostra`, utilizamos uma `?` no lugar dos valores e passamos um `array` na ordem em que esses elementos devem ser substituídos. Não tem muito segredo, mas se você não está muito acostumado com SQL pode se assustar um pouco. Muito em breve veremos como o Laravel nos ajuda neste trabalho.

8.4 Retornando uma view de confirmação

Agora que já estamos persistindo, podemos retornar uma *view* com a mensagem de confirmação, dizendo que o produto foi adicionado com sucesso. Podemos chamá-la de `adicionado.blade.php`.

```
@extends('templates.principal')

@section('conteudo')

O produto foi adicionado com sucesso!

@stop
```

Vale lembrar que essa view será criada dentro de `/resources/views/produto`, assim como as demais, portanto, no retorno do controller faremos `produto.adicionado`:

```
public function adiciona(){

  $nome = Request::input('nome');
```

```
  $valor = Request::input('valor');
  $descricao = Request::input('descricao');
  $quantidade = Request::input('quantidade');

  DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));

  return view('produto.adicionado');
}
```

Já podemos fazer um novo teste, garantindo que a mensagem será exibida e que o produto será persistido no banco.

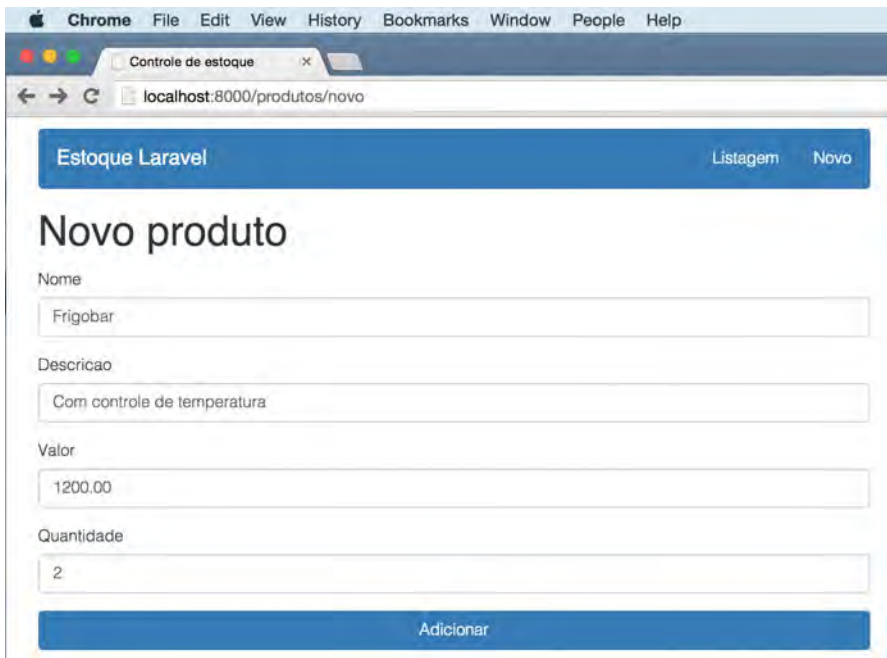



Fig. 8.5: Adicionando um novo produto.

Ao submeter o formulário, recebemos o seguinte resultado:

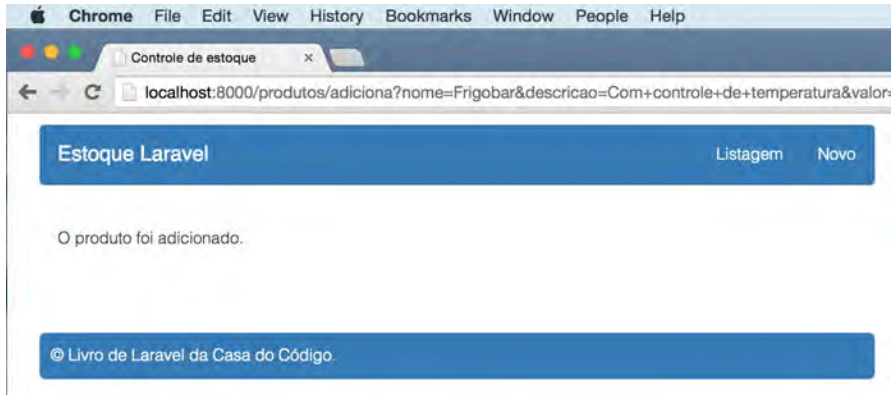



Fig. 8.6: Mensagem de produto adicionado em texto puro.

Tudo funcionou como esperado, mas ainda assim alguns detalhes podem ser melhorados. Seria legal, por exemplo, exibir o nome do produto que foi adicionado, não é? Isso é bem simples, basta incluir o parâmetro pelo controller, com o método `with` que já vimos. O método `adiciona` completo ficará como o seguinte:

```
public function adiciona(){

  $nome = Request::input('nome');
  $valor = Request::input('valor');
  $descricao = Request::input('descricao');
  $quantidade = Request::input('quantidade');

  DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));

  return view('produto.adicionado')->with('nome', $nome);
}
```

Na view, exibimos o `$nome` incluído:

```
@extends('templates.principal')
```

```
@section('conteudo')

O produto {{$nome}} foi adicionado com sucesso!

@stop
```

Além disso, podemos adicionar uma *div* com as classes do bootstrap para deixar a mensagem com uma aparência um pouco mais amigável. Uma alternativa seria:

```
@extends('templates.principal')

@section('conteudo')

<div class="alert alert-success">
    <strong>Sucesso!</strong> O produto {{$nome}} foi adicionado.
</div>

@stop
```

Agora sim, o resultado será:

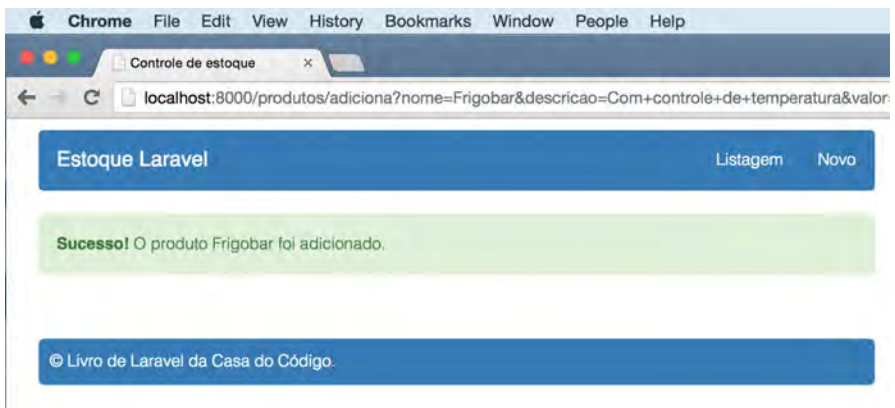



Fig. 8.7: Mensagem de produto adicionado com estilos do bootstrap.

Vamos acessar também a listagem, para garantir que o produto foi persistido no banco de dados como deveria.

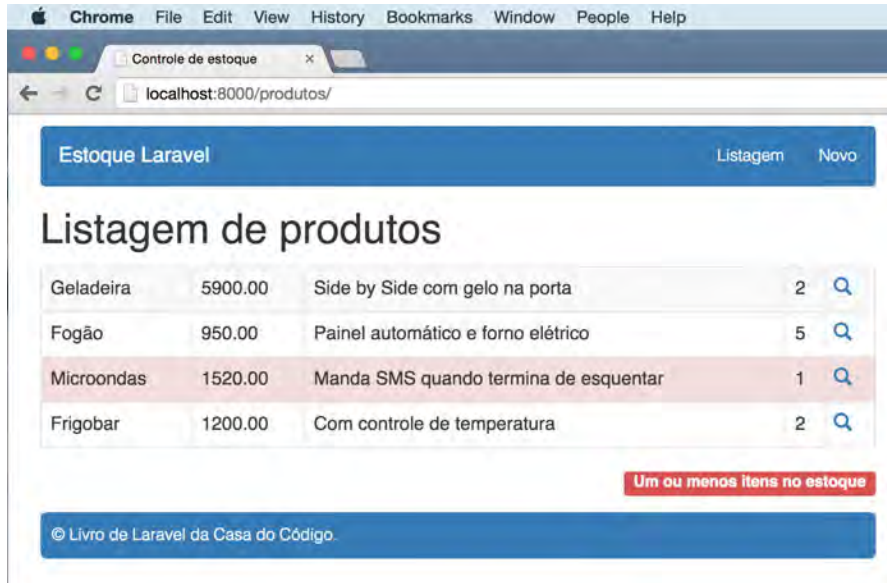



Fig. 8.8: Listagem já com novo produto adicionado.

Perfeito, deu tudo certo!

Query Builder

Em vez de utilizar o `DB:insert` como fizemos, temos como alternativa um recurso conhecido como *query builder*. Ele nos oferece uma interface fluente, que é bastante conveniente para não nos preocuparmos tanto com o SQL. A adição de produtos poderia ficar assim:

```
DB::table('produtos')->insert(
    ['nome' => $nome,
      'valor' => $valor,
      'descricao' => $descricao,
      'quantidade' => $quantidade
    ]
);
```

Legal, não é? Se você se interessar, pode ver mais das possibilidades em http://laravel.com/docs/4.0/queries. Em breve veremos uma forma ainda mais simples de fazer esse trabalho, com uso do `Eloquent`.

8.5 Utilizando os métodos do HTTP

O cadastro de produtos está funcionando bem, ele é adicionado ao banco, uma página de confirmação bonita é exibida, tudo parece correto. Só que há um ponto muito importante que precisa ser corrigido.

Veja que, ao adicionar um produto, todos os parâmetros da requisição estão sendo enviados pela URL:

```
http://localhost:8000/produtos
  /adiciona?nome=Frigobar
  &descricao=Com+controle+de+temperatura
  &valor=1200.00&quantidade=2
```

Isso acontece porque, por padrão, o método HTTP utilizado no `form` do HTML é o `GET`. Mas nem sempre queremos deixar tudo visível dessa forma, imagine por exemplo se esse fosse um formulário de login, a URL ficaria assim:

```
http://localhost:8000
  /login?usuario=Rodrigo
  &senha=mamae+querida
```

Um caos! Além disso, existe um limite para o tamanho da URL, ou seja, se fosse um formulário muito grande, ou pequeno com muito texto, correríamos o risco de perder informações se esse conteúdo ultrapassasse esse limite.

Em vez de enviar por `GET`, podemos modificar o formulário para utilizar um outro método HTTP, o `POST`. Dessa forma, os dados não ficarão mais visíveis na URL, tudo será passado dentro do corpo do protocolo HTTP.

A URL ficará http://localhost:8000/produtos/adiciona, sem os dados visíveis, independente da quantidade de parâmetros enviados. É muito importante lembrar que isso não é uma medida de segurança, afinal os dados ainda podem ser consultados pelo navegador do usuário. O ganho dessa abordagem é que o corpo do `POST` é muito maior que a URI do `GET`, e evitamos deixar os dados sempre tão expostos.

Tudo o que precisamos fazer para que os dados sejam enviados via `POST` no *form* de novo de produto é adicionar o atributo `method="post"`, como a seguir:

```
<h1>Novo produto</h1>

<form action="/produtos/adiciona" method="post">
  <div class="form-group">
    <label>Nome</label>
    <input name="nome" class="form-control"/>
  </div>

<!-- continuação do html -->
```

Mas, com essa alteração, nosso formulário para de funcionar! Tente adicionar um novo produto para ver o resultado.

Fig. 8.9: MethodNotAllowedHttpException ao submeter o formulário.

Recebemos uma `MethodNotAllowedHttpException`, mas o que isso significa? O que acontece é que no momento de mapear uma rota para o método `adiciona`, fizemos:

```
Route::get('/produtos/adiciona', 'ProdutoController@adiciona');
```

Repare que, como utilizamos o método `get`, definimos explicitamente que apenas requisições desse tipo serão aceitas. É justo. Como resolver? Uma possibilidade seria utilizando o método `any`, ou seja, liberando qualquer tipo de requisição para essa URL. Mas não é isso que queremos, não é? Queremos enviar os dados por `POST`; portanto, utilizaremos o método `post`:

```
Route::post('/produtos/adiciona', 'ProdutoController@adiciona');
```

Para saber mais: O método match

Além de `get`, `post` e `any`, a interface `Route` ainda nos oferece um método chamado `match`, que nos permite passar um `array` especificando exatamente quais métodos HTTP devem ser aceitos. Um exemplo seria:

```
Route::match(array('GET', 'POST'),
  '/produtos/adiciona',
  'ProdutoController@adiciona');
```

Neste caso, requisições de tipo `GET` ou `POST` seriam aceitas.

Vamos testar? É só abrir o formulário novamente no navegador, preencher os campos e clicar em adicionar. Para nossa surpresa, ou não, ainda não funcionará. Mas desta vez o erro será outro:

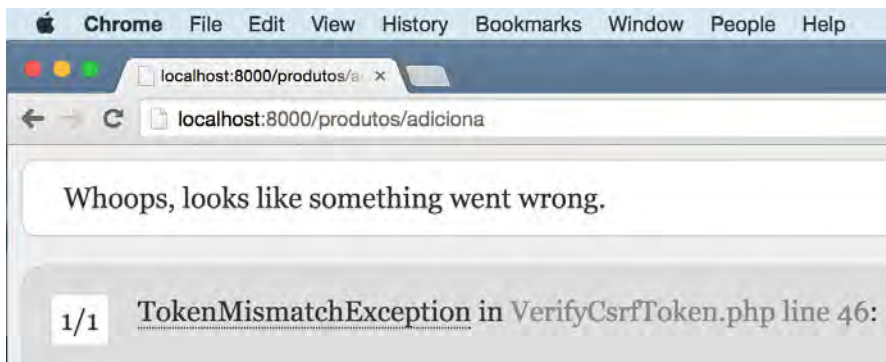



Fig. 8.10: TokenMismatchException ao submeter o formulário.

Recebemos um `TokenMismatchException`, que acontece sempre que enviamos uma requisição do tipo `POST` sem mandar um parâmetro chamado `_token`. Mas por que isso é necessário? Por padrão, o Laravel tem um mecanismo de proteção contra **CSRF** (*Cross-site request forgery*, ou falsificação de solicitação entre sites). Esse é um tipo de ataque, bastante conhecido no

mundo web, que tira proveito da relação de confiança entre seu usuário legítimo e um aplicativo web. Você pode ler um pouco mais sobre ele aqui:

http://pt.wikipedia.org/wiki/Cross-site_request_forgery

A solução é simples, o Laravel tem um *helper method* chamado `csrf_token()` que nos retorna esse token. Só precisamos adicionar um novo *input*, que pode e deve ficar escondido, portanto será `type="hidden"`. Além disso, o `name` desse input precisa ser `_token`, que é esperado pelas requisições desse tipo. O *input* ficará assim:

```
<input type="hidden"
  name="_token" value="{{{ csrf_token() }}}" />
```

O código no nosso `formulario.blade.php` completo fica:

```
@extends('layout.principal')

@section('conteudo')

<h1>Novo produto</h1>

<form action="/produtos/adiciona" method="post">

  <input type="hidden"
    name="_token" value="{{{ csrf_token() }}}" />

  <div class="form-group">
    <label>Nome</label>
    <input name="nome" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Descricao</label>
    <input name="descricao" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Valor</label>
    <input name="valor" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Quantidade</label>
```

```
    <input type="number"
      name="quantidade" class="form-control"/>
  </div>
  <button type="submit"
    class="btn btn-primary btn-block">Adicionar</button>
</form>

@stop
```

Agora sim, vamos testar?

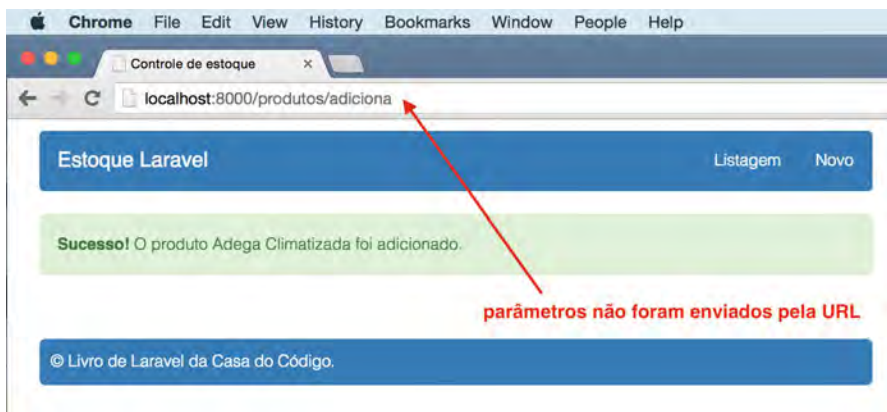



Fig. 8.11: Formulário enviado com sucesso.

Sucesso!

8.6 Mais métodos HTTP e quando utilizá-los

Mas, afinal, quando devemos usar cada método HTTP? É muito comum utilizarmos o método `GET` para listagens ou visualização de algum de nossos recursos, sempre em operações que **não modificam os valores do sistema**.

O `POST` deve ser utilizado para adição de informações a um recuso existente ou **adição de um novo recurso**, como estamos fazendo agora com os produtos.

Além disso, existem outros verbos como o `PUT` e `DELETE`, que são utilizados para atualização e remoção, respectivamente.

Capítulo 9

Os diferentes tipos de resposta

O problema da abordagem que utilizamos ao adicionar um novo produto é que, normalmente, não queremos ir para uma tela que apenas mostra uma mensagem. Se pensarmos bem, a view `adicionado.blade.php` não tem muita utilidade. A mensagem que ela exibe, sim, é muito válida, mas isso pode ser feito de diversas outras formas, sem a necessidade de uma view intermediaria.

9.1 Redirecionando para outras lógicas

Uma alternativa bastante utilizada seria encaminhar o usuário para a página de listagem após a adição de um novo produto. Podemos ainda exibir a mensagem de que o produto foi adicionado com sucesso na própria listagem, como é de costume. Mas o que precisaríamos alterar em nosso código?

Em vez de retornar a view `adicionada`, como estamos fazendo, po-

demos modificar o final do método `adiciona` para que ele retorne a view `listagem`.

```
public function adiciona(){

  $nome = Request::input('nome');
  $valor = Request::input('valor');
  $descricao = Request::input('descricao');
  $quantidade = Request::input('quantidade');

  DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));

  return view('produtos.listagem');
}
```

Mas isso não seria o suficiente, afinal, a listagem precisa da variável `$produtos` com todos os registros dessa tabela no banco de dados para funcionar. Em outras palavras, com essa alteração, a view de listagem será exibida corretamente, porém ela sempre estará vazia.

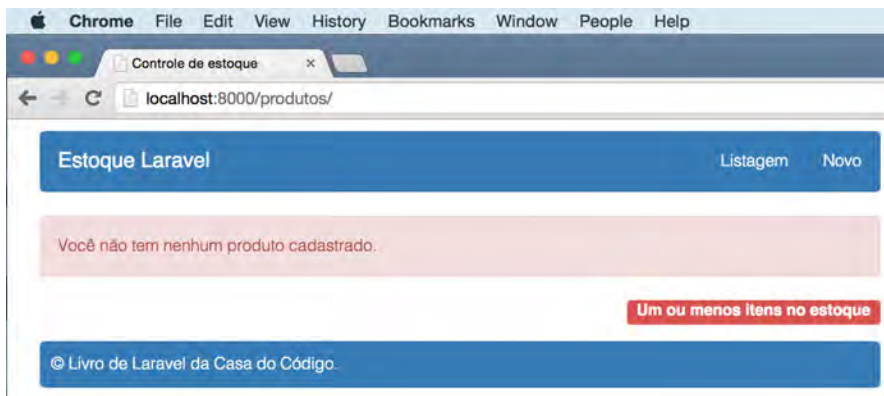



Fig. 9.1: Listagem de produtos vazia.

Bem, não é o que esperávamos! Mas a solução é bem simples, basta listar os produtos e passar a variável para view, por exemplo:

```
public function adiciona(){

  $nome = Request::input('nome');
  $valor = Request::input('valor');
  $descricao = Request::input('descricao');
  $quantidade = Request::input('quantidade');

  DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));

  $produtos = DB::select('select * from produtos');
  return view('produtos.listagem')->with('produtos', $produtos);
}
```

Agora sim, tudo vai funcionar como esperado! Mas calma, o final do nosso método `adiciona` ficou idêntico ao método `lista`:

```
public function lista(){
  $produtos = DB::select('select * from produtos');
  return view('produtos.listagem')->with('produtos', $produtos);
}
```

Sabemos que isso não é legal, afinal, estamos duplicando código. Do ponto de vista da orientação a objetos, isso é um problema bem grande, já que não estamos seguindo seu princípio básico de reutilizar comportamentos. Se o método `lista` já faz esse trabalho, não queremos copiar seu código, mas sim delegar essa responsabilidade para ele.

A mudança no método `adiciona` será mínima, mas muito valiosa. Em vez de fazer o `select` no banco e retornar a view de listagem, simplesmente redirecionamos o fluxo para a URI `/produtos`, que será atendida pelo método `lista`. Assim como o `view`, há um *helper method* que nos ajuda a fazer redirecionamentos como esse:

```
return redirect('/produtos');
```

Simples, não é? Podemos e devemos redirecionar o fluxo da requisição para outras lógicas sempre que necessário. O método `adiciona` completo ficará assim:

```
public function adiciona(){

    $nome = Request::input('nome');
    $valor = Request::input('valor');
    $descricao = Request::input('descricao');
    $quantidade = Request::input('quantidade');

    DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));

    return redirect('/produtos');
}
```

Ainda há muito o que melhorar, mas essa alteração já foi um passo para o caminho certo. Teste para garantir que tudo está funcionando como esperado, após adicionar um novo produto, a tela de listagem deve ser exibida.

9.2 Mas e a mensagem de confirmação?

O único problema dessa alteração é que agora não temos mais uma mensagem bonita com o feedback de que o processo foi concluído com sucesso, mas isso é simples, podemos mover a *div* da página `adicionado.blade.php` para o final da listagem:

```
@extends('layout.principal')

@section('conteudo')

 @if(empty($produtos))
  <div class="alert alert-danger">
    Você não tem nenhum produto cadastrado.
  </div>

 @else
  <h1>Listagem de produtos</h1>
  <table class="table table-striped ...">
    @foreach ($produtos as $p)
```

```
        <!-- conteúdo omitido -->
    @endforeach
  </table>
 @endif

<h4>
    <span class="label label-danger pull-right">
        Um ou menos itens no estoque
    </span>
</h4>

<div class="alert alert-success">
<strong>Sucesso!</strong>
    O produto {{ $nome }} foi adicionado.
</div>

@stop
```

Vamos testar? Adicionamos um novo produto, mas veja o resultado:

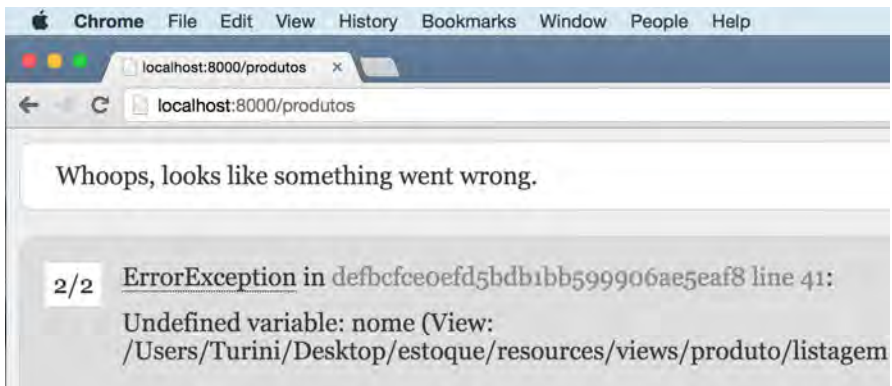



Fig. 9.2: Exception de variável nome não definida.

Uma exception! O `Undefined variable: nome`, ou variável `nome` não definida em bom português, aconteceu porque, quando fizemos o *redirect*, os dados da requisição anterior foram perdidos. Isso é natural, claro, afinal o esperado é que os dados de uma requisição morram junto com ele,

mas como resolver? Há algumas alternativas, uma das quais seria incluir o parâmetro manualmente na nova requisição feita pelo *redirect*, mas e se forem vários? Daria um bom trabalho!

9.3 Recuperando valores da requisição anterior

Por nossa sorte o Laravel nos oferece uma forma bem simples de recuperar os valores da requisição anterior, já que é uma necessidade comum em casos como esse. Observe como é simples:

```
public function adiciona(){

    $nome = Request::input('nome');
    <!-- código omitido -->

    return redirect('/produtos')->withInput();
}
```

É só isso, basta chamar o método `withInput` depois do `redirect` e pronto, todos os parâmetros serão mantidos. Mas há uma questão. Na view, em vez de acessar o parâmetro da forma tradicional, utilizamos um método auxiliar chamado `old`. Portanto, o código do final da listagem deverá ficar assim:

```
<div class="alert alert-success">
    <strong>Sucesso!</strong>
        O produto {{ old('nome') }} foi adicionado.
</div>
```

Para mim é uma abordagem bem justa, pois dessa forma fica explícito que estamos exibindo um valor da requisição anterior. Vamos testar? Tentamos adicionar um novo produto e sucesso, a mensagem foi exibida:

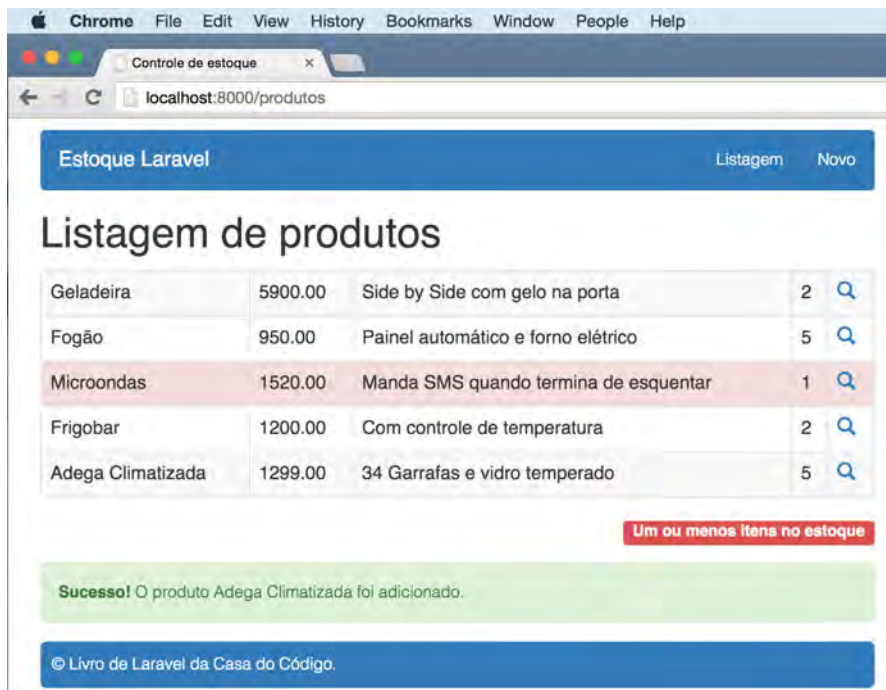



Fig. 9.3: Listagem com mensagem de produto adicionado com sucesso.

Mas vamos fazer um novo teste, abrindo a listagem sem que nenhum produto tenha sido adicionado. Ops, a mensagem está lá!

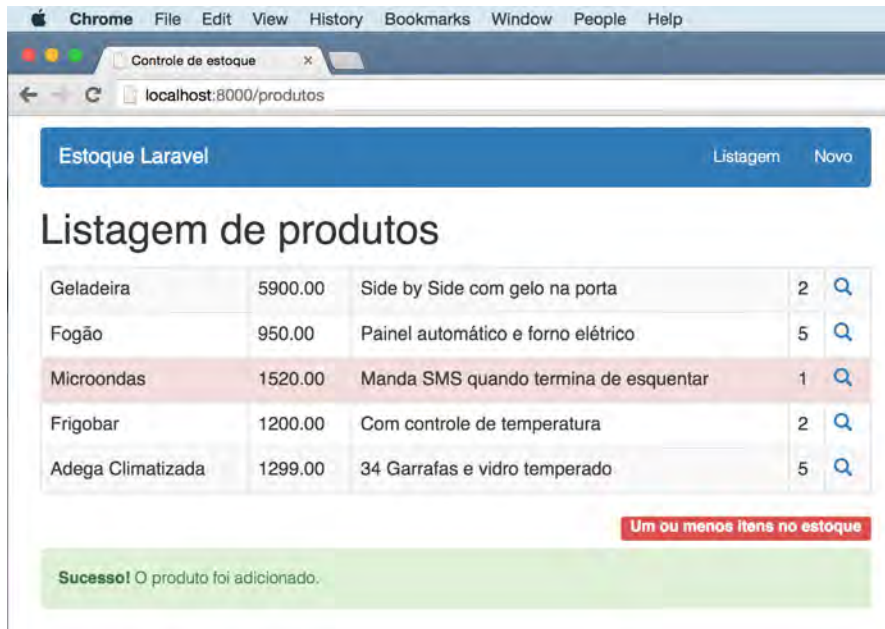



Fig. 9.4: Mensagem de sucesso sem nome de produto.

Veja que a única diferença é que o nome ficou em branco, já que nenhum produto foi adicionado. Mas a solução é bem simples, só precisamos garantir que a mensagem será exibida **apenas se** o valor do nome estiver presente na requisição anterior. Um simples `if` resolve o problema:

```
@if(old('nome'))
  <div class="alert alert-success">
    <strong>Sucesso!</strong>
        O produto {{ old('nome') }} foi adicionado.
  </div>
@endif
```

9.4 Escolhendo quais valores manter

O problema de usar o `withInput` dessa forma é que todos os parâmetros são mantidos, mas isso é desnecessário, já que estamos usando apenas o **nome**.

Para evitar que isso aconteça, você pode definir explicitamente quais parâmetros devem ser mantidos:

```
return redirect('/produtos')
    ->withInput(Request::only('nome'));
```

Legal, não é? O mesmo vale para os outros métodos da `Request`, como por exemplo o `except`. No caso a seguir, todos os parâmetros exceto a senha serão mantidos na próxima requisição.

```
return redirect('/usuarios')
    ->withInput(Request::except('senha'));
```

9.5 Outros tipos de redirect

Além de como fizemos, existem diversas possibilidades interessantes para o método `redirect`. Uma delas, que por sinal eu recomendo bastante, é redirecionar para uma ação do controller e não uma URI.

Redirecionando para uma action

Quer ver como é simples? Como o método `adiciona` quer redirecionar para o método `lista`, da classe `ProdutoController`, em vez de usar o `redirect('/produtos')`, ou seja, com a URI, ele pode fazer:

```
return redirect()
    ->action('ProdutoController@lista')
    ->withInput(Request::only('nome'));
```

Assim ele está dizendo **explicitamente qual o método que quer redirecionar**, independente de sua URI. Se decidirmos no futuro mudar a rota desse método, precisaríamos lembrar de mudar todos os lugares que faziam `redirect`. Já quando fazemos dessa forma, apontando para uma `action` específica, tudo continua funcionado.

Claro, se alguém mudar o nome do método o `redirect` também vai parar de funcionar, mas esse tipo de mudança acontece com uma frequência bem menor do que a mudança de URI.

Mas oferecer essa possibilidade apenas para o `redirect` não ajudaria muito, pois, todos os links quebrariam se uma rota fosse alterada! Imagine que em vez de `/produtos` a URI da listagem fosse alterada para `/produtos/lista`. O que acontece quando você clicar no link de listagem do menu? Quebra!

É justo, afinal, ele vai continuar apontando para a URI anterior já que estamos deixando isso fixo (ou *hardcoded*, como é comumente chamado) em nosso HTML. Repare no *navbar* do layout `principal.blade.php`:

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="/produtos">Listagem</a></li>
  <li><a href="/produtos/novo">Novo</a></li>
</ul>
```

9.6 Para saber mais: rotas nomeadas

O Laravel ainda oferece uma outra alternativa para o `redirect`. Em vez de redirecionar para uma *URI* ou *action*, é possível definir um nome para suas rotas, ao declará-las no arquivo `routes.php`. Um exemplo seria:

```
Route::get('/produtos', [
    'as' => 'apelido',
    'uses' => 'ProdutoController@lista'
]);
```

Esse recurso é conhecido como *named route*. Uma vez que tenhamos definido um apelido para a rota, podemos passar a fazer os *redirects* da seguinte forma:

```
return redirect()->route('apelido');
```

Linkando para ações

Em vez de linkar para uma URI, é sempre mais interessante linkar para uma ação do controller, assim como fizemos no *redirect*. A mudança será simples, basta chamar o método auxiliar `action` de dentro das chaves duplas do blade. Os links devem ficar assim:

```
<ul class="nav navbar-nav navbar-right">
  <li>
      <a href="{{action('ProdutoController@lista')}}">
          Listagem
      </a>
  </li>
  <li>
      <a href="{{action('ProdutoController@novo')}}">
          Novo
      </a>
  </li>
</ul>
```

Faça a alteração e clique nos links para testar, o resultado será o mesmo! Se você clicar com o botão direito no navegador e escolher a opção *view page source*, verá que o código-fonte do HTML gerado continuará igual:

Fig. 9.5: Código HTML da listagem, com os links após uso do action.

9.7 Outros tipos de resposta

Além de retornar uma view, ou redirecionar para outra lógica, existem momentos em que queremos enviar outros tipos de resposta, por exemplo quando estamos trabalhando com serviços ou comunicação entre sistemas.

Um formato bem comum e muito utilizado atualmente é o JSON, e por esse motivo ele é o formato padrão de resposta do Laravel. Você em algum momento já experimentou retornar um objeto ou variável em vez de uma view? Se não fez, com certeza vai gostar de testar.

Podemos criar um novo método no `ProdutoController` chamado `listaJson`, que pode fazer o seguinte:

```
public function listaJson(){
    $produtos = DB::select('select * from produtos');
    return $produtos;
}
```

Não se esqueça de que a cada novo método uma rota deve ser criada no `routes.php`, para que ele fique acessível pelo navegador.

```
Route::get('/produtos/json', 'ProdutoController@listaJson');
```

Excelente, agora que já temos o método e ele está devidamente mapeado, podemos acessar sua URL pelo navegador:

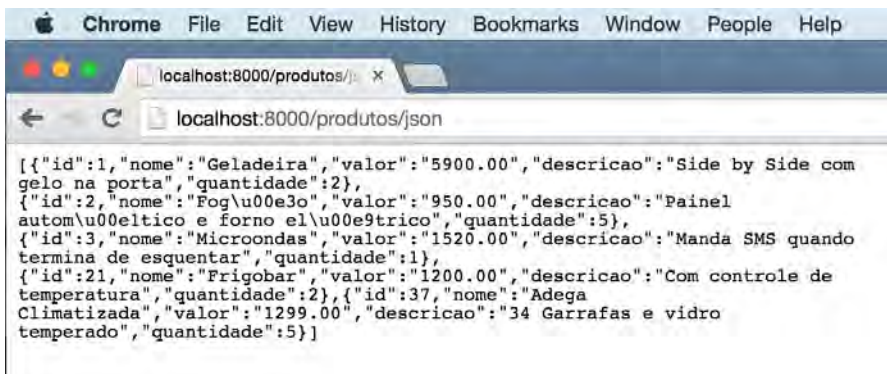
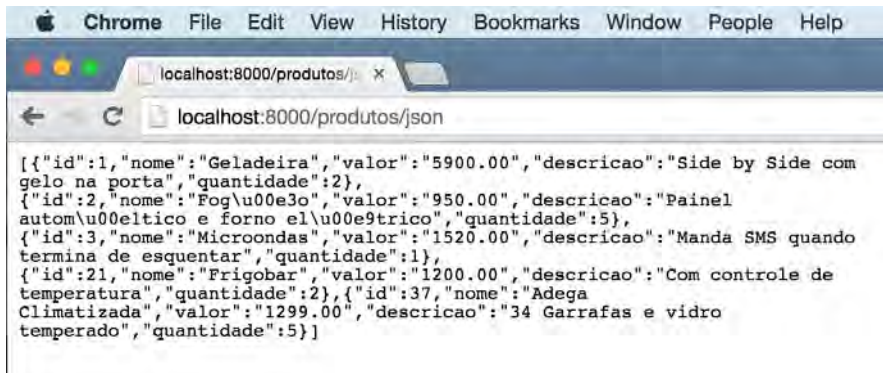



Fig. 9.6: Lista de produtos em formato JSON.

O resultado foram todos os produtos *serializados* em formato `json`. Bem simples, não foi? Mas também podemos fazer isso explicitamente. Em vez de retornar a lista de produtos, poderíamos fazer:

```
public function listaJson(){
    $produtos = DB::select('select * from produtos');
    return response()->json($produtos);
}
```

Executando no navegador, o resultado será o mesmo!

Fig. 9.7: Lista de produtos em formato JSON.

Claro, no caso de o JSON retornar o objeto ou lista de objetos diretamente seria muito mais simples, mas além do método `json` existem diversos outros que podem ser muito úteis em nosso dia a dia. Um exemplo seria:

```
return response()
    ->download($caminhoParaUmArquivo);
```

Acessar um método com esse retorno resultaria no download do arquivo presente no caminho especificado. Muito simples e prático, não acha?

Capítulo 10

Eloquent ORM

Nossa aplicação está cada vez mais completa, porém os métodos do controller estão ficando mais complexos, com muitas linhas de código, e exigem um certo conhecimento de SQL. Veja o método `adiciona`, por exemplo:

```
public function adiciona(){

    $nome = Request::input('nome');
    $valor = Request::input('valor');
    $descricao = Request::input('descricao');
    $quantidade = Request::input('quantidade');

    DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));
```

```
    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

A ordem em que os parâmetros são passados para o método `insert` é importantíssima! Um erro pode causar uma confusão em nosso banco de dados. Além disso, se você não está acostumado com SQL, deve estar achando bastante complicado, não é?

Neste capítulo veremos como o Laravel, com ajuda de outro framework, pode nos auxiliar nesses problemas. O método `adiciona`, por exemplo, terminará com 2 linhas, muito mais legível e fácil de manter.

10.1 Conheça a solução: ORM

Sabemos que o problema dessa abordagem é que estamos nos preocupando muito com instruções SQL, o código fica verboso e exige um conhecimento maior sobre outro paradigma com o qual, na maior parte do tempo, não queremos ficar nos preocupando.

Há uma forma bem mais simples e interessante de fazer essa e outras operações no banco de dados, sem termos que nos preocupar tanto com SQL e detalhes do mundo relacional dos bancos de dados. Integrado ao Laravel temos outro framework, conhecido como **Eloquent ORM**. Ele cuida de encapsular boa parte da complexidade do mundo relacional, permitindo acesso aos nossos dados de uma forma bem elegante e fluente.

Frameworks ORM

Por mais que o PHP já simplifique muito esse processo, ainda assim gastamos uma boa parte do nosso desenvolvimento com *queries* SQL e nos preocupando com detalhes de seu paradigma. Quer trocar de banco? Nem sempre isso será fácil. Apesar do padrão, muita coisa muda de acordo com o fabricante. Há ainda o desafio de pensar de duas maneiras diferentes, já que o paradigma relacional é bem diferente da orientação a objetos.

Essas são algumas das muitas razões que motivaram os frameworks **ORM** (***O**bject **R**elational **M**apping*), que como o próprio nome indica, fazem o mapeamento dos nossos objetos para o mundo relacional dos bancos de dados. Você vai perceber que essa abstração é bastante conveniente, por isso é tão bem-vinda em linguagens como o PHP, Java, C# e outras.

10.2 Tudo mais simples com Eloquent

Quer ver como é fácil começar a usar o Eloquent? Em vez de sair escrevendo SQLs, só precisamos usar a classe de **model**o que criamos anteriormente para representar nossa tabela de produtos.

Veja que, assim como qualquer classe de modelo do Eloquent, a classe `Produto` herda de `Model`. Toda classe que herda de `Model` ganha uma série de métodos que nos auxilia a fazer operações no banco de dados. O primeiro método que usaremos será o `all`, que retorna todos os registros da tabela.

Veja como estamos fazendo isso no método `lista` atualmente:

```
public function lista(){
  $produtos = DB::select('select * from produtos');
  return view('produtos.listagem')->with('produtos', $produtos);
}
```

No lugar disso, com método `all`, o código ficará assim:

```
public function lista(){
  $produtos = Produto::all();
  return view('produtos.listagem')->with('produtos', $produtos);
}
```

O resultado será o mesmo, só que agora não existe nenhum instrução SQL nesse método. Resolvemos o problema com uma simples chamada de método. Vamos testar?

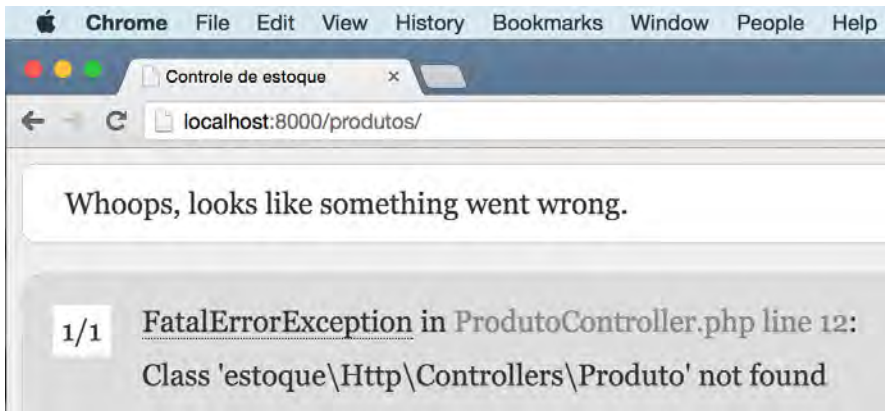



Fig. 10.1: Exception de classe Produto não encontrada.

Ops, precisamos importar a classe `Produto`!

```
<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;
use estoque\Produto;
use Request;

class ProdutoController extends Controller {

// restante do código escondido
```

Agora sim, vamos abrir a listagem novamente:

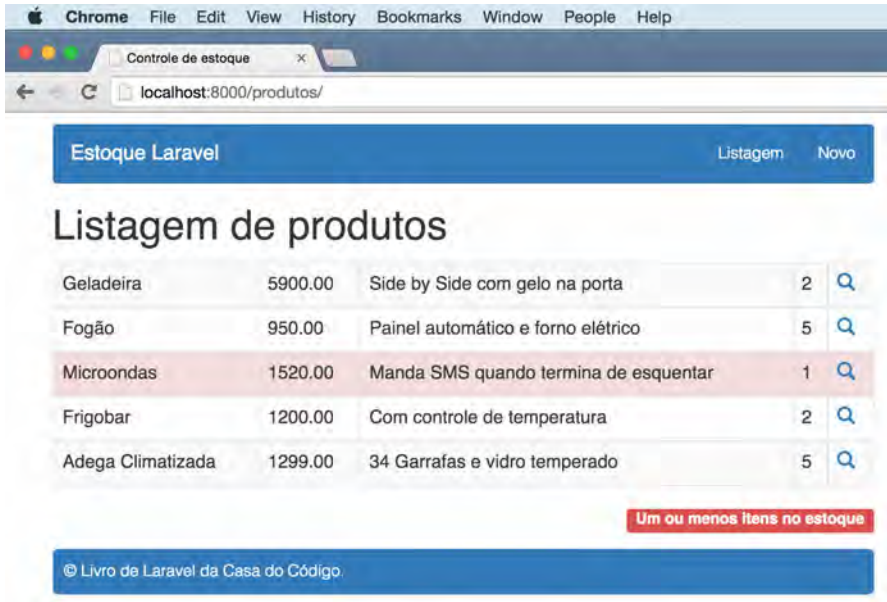



Fig. 10.2: Listagem de produtos.

Sucesso, tudo funcionando. Podemos fazer o mesmo com o método `listaJson`, que atualmente está assim:

```
public function listaJson(){
    $produtos = DB::select('select * from produtos');
    return response()->json($produtos);
}
```

Agora, com o método `all`, ele pode ficar assim:

```
public function listaJson(){
    $produtos = Produto::all();
    return response()->json($produtos);
}
```

10.3 Buscando pelo ID com Eloquent

Outro ponto em que podemos tirar proveito do Eloquent é na busca de produtos pelo `id`, como é feito no método `mostra`:

```
public function mostra($id){
    $resposta =
        DB::select('select * from produtos where id = ?', [$id]);
    if(empty($resposta)) {
        return "Esse produto não existe";
    }
    return view('produto.detalhes')->with('p', $resposta[0]);
}
```

Para isso funcionar foi necessário passarmos um array com os parâmetros pro método `select`, que além disso retorna um array. Muito trabalhoso, não é? Que tal trocar todo esse trabalho por um simples `find($id)`?

```
public function mostra($id){
    $produto = Produto::find($id);
    if(empty($produto)) {
        return "Esse produto não existe";
    }
    return view('produto.detalhes')->with('p', $produto);
}
```

Com isso temos um código mais simples, legível e expressivo. Não se esqueça de testar:

Fig. 10.3: Detalhes do produto.

10.4 Migrando a inserção de produtos

O método `adiciona` é um dos mais problemáticos, e com isso, o ganho será ainda maior. Lembra como ele está? Dê uma boa olhada, esse código todo vai sumir em breve!

```
public function adiciona(){

    $nome = Request::input('nome');
    $valor = Request::input('valor');
    $descricao = Request::input('descricao');
    $quantidade = Request::input('quantidade');

    DB::insert('insert into produtos
    (nome, quantidade, valor, descricao) values (?,?,?,?)',
    array($nome, $valor, $descricao, $quantidade));

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Vamos melhorá-lo aos poucos, a começar, tirando esse SQL daí:

```
public function adiciona(){

    $produto = new Produto();
    $produto->nome = Request::input('nome');
    $produto->valor = Request::input('valor');
    $produto->descricao = Request::input('descricao');
    $produto->quantidade = Request::input('quantidade');

    $produto->save();

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

A diferença é que agora instanciamos um produto, populamos os valores da requisição e, em vez de executar um SQL, simplesmente chamamos o mé-

todo save. Bem melhor, mas ainda tem muito código aí, não é? Podemos reduzir ainda mais:

```
public function adiciona(){

    $params = Request::all();
    $produto = new Produto($params);
    $produto->save();

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Note que estamos passando o array com os parâmetros da requisição no construtor do modelo nesse caso Produto. Isso é muito comum no mundo PHP e de outras linguagens, mas pode ser um pouco perigoso. Por isso, ao tentar adicionar um produto após essa alteração, o resultado será:

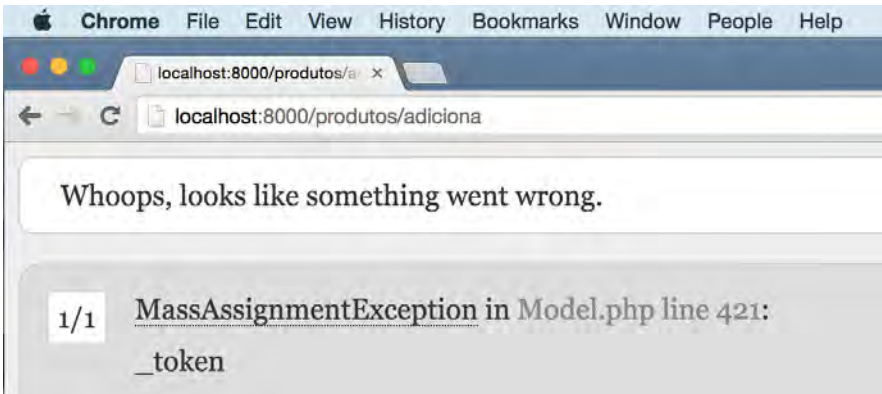



Fig. 10.4: MassAssignmentException ao tentar adicionar.

Recebemos um MassAssignmentException, mas o que isso significa?
::Mass-assignment

Sendo assim, sempre que quisermos fazer a atribuição dessa forma, via *mass-assignable*, para nos proteger, precisamos adicionar uma propriedade chamada $fillable em nosso modelo especificando exatamente

quais atributos podem ser populados, caso contrário receberemos uma `MassAssignmentException`. No caso do produto, serão o nome, descrição, valor e quantidade:

```
protected $fillable = array('nome',
    'descricao', 'valor', 'quantidade');
```

A classe completa deve ficar assim:

```
<?php namespace estoque;

use Illuminate\Database\Eloquent\Model;

class Produto extends Model {

    protected $table = 'produtos';
    public $timestamps = false;

    protected $fillable = array('nome',
        'descricao', 'valor', 'quantidade');
}
```

Tudo pronto, agora a adição voltará a funcionar como esperado.

Uma outra opção: guarded

Além do `$fillable`, você também pode declarar um atributo chamado `$guarded`, que faz justamente o papel inverso. Ele é uma espécie de *black-list* dos valores cuja atribuição você não quer permitir via *mass-assignment*. Um bom caso de uso para o `$guarded` seria impedir que o usuário altere o `id` de seu modelo. Veja como é simples:

```
<?php namespace estoque;

use Illuminate\Database\Eloquent\Model;

class Produto extends Model {

    protected $table = 'produtos';
    public $timestamps = false;

    protected $fillable = array('nome',
        'descricao', 'valor', 'quantidade');

    protected $guarded = ['id'];
}
```

O código ficou bem mais simples, não é?

```
public function adiciona(){

    $params = Request::all();
    $produto = new Produto($params);
    $produto->save();

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Mas, apesar de já estar simples, podemos enxugá-lo ainda mais. Em vez de criar uma nova instância de produto, podemos utilizar um *factory method* chamado `create`:

```
$params = Request::all();
Produto::create($params);
```

Nem precisamos chamar o método `save`! Legal, não é? Agora que está mais simples, podemos declarar o código em um único *statement*:

```
Produto::create(Request::all());
```

Veja como nosso código final ficou mais enxuto:

```
public function adiciona(){

    Produto::create(Request::all());

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

10.5 Função de remover produtos

Para deixar nossa aplicação ainda mais completa e de bônus conhecer um novo método do Eloquent, que tal criarmos uma nova função de remover produtos? O fundamental nós já sabemos muito bem, criamos uma nova rota no `routes.php`:

```
Route::get('/produtos/remove/{id}', 'ProdutoController@remove');
```

Note que, assim como no método `mostra`, estamos passando o `id` como *path param*, sendo assim podemos recebê-lo como argumento no método `remove`, que vamos criar no `ProdutoController`:

```
public function remove($id){
    // ...
}
```

Assim como as outras operações, deletar um produto com Eloquent é bem simples. Só precisamos buscar o modelo que deve ser excluído e chamar seu método `delete`:

```
public function remove($id){
    $produto = Produto::find($id);
    $produto->delete();
}
```

Excelente, e ao final, podemos redirecionar para a listagem:

```
public function remove($id){
    $produto = Produto::find($id);
    $produto->delete();
    return redirect()
        ->action('ProdutoController@lista');
}
```

Quase tudo pronto, só precisamos adicionar um novo link na tabela da view `listagem.blade.php`. Ele ficará muito parecido com o link de exibir detalhes do produto, mudando apenas seu ícone e a *action* que será chamada:

```
<table class="table table-striped ...">
@foreach ($produtos as $p)
  <!-- código escondido -->
  <td>
    <a href="{{action('ProdutoController@remove', $p->id)}}">
      <span class="glyphicon glyphicon-trash"></span>
    </a>
  </td>
 </tr>
@endforeach
</table>
```

Vamos ver o resultado? Ao recarregar a página de listagem de produtos no navegador, o novo ícone com link estará disponível:

Fig. 10.5: Listagem com icone de remover.

Tente excluir um produto para testar!

10.6 Código final do ProdutoController

Após essa série de mudanças com a introdução do Eloquent, o código do nosso controller de produtos ficou assim:

```
<?php namespace estoque\Http\Controllers;

use estoque\Produto;
use Request;
use estoque\Http\Requests\ProdutosRequest;

class ProdutoController extends Controller {

    public function lista(){
        $produtos = Produto::all();
        return view('produto.listagem')
            ->with('produtos', $produtos);
    }

    public function mostra($id){
        $produto = Produto::find($id);
```

```
        if(empty($produto)) {
            return "Esse produto não existe";
        }
        return view('produto.detalhes')
            ->with('p', $produto);
    }

    public function novo(){
        return view('produto.formulario');
    }

    public function adiciona(){

        Produto::create(Request::all());

        return redirect()
            ->action('ProdutoController@lista')
            ->withInput(Request::only('nome'));
    }

    public function listaJson(){
        $produtos = Produto::all();
        return response()->json($produtos);
    }

    public function remove($id){
        $produto = Produto::find($id);
        $produto->delete();
        return redirect()
            ->action('ProdutoController@lista');
    }
}
```

Mais simples, não acha? O Eloquent abstrai boa parte da complexidade de se trabalhar com instruções do banco de dados. Antes de seguir, que tal dar uma olhada na documentação desse framework? Com certeza você encontrará muita coisa legal por lá.

http://laravel.com/docs/eloquent

Desafio

Que tal tentar implementar a função de alterar produtos por conta própria? Esse é o momento em que você perceberá se realmente está afiado no Laravel. O método de alterar e seu formulário HTML serão muito parecidos com o de adicionar, com a diferença de que você precisará passar o **id** do produto como parâmetro. Ficou com alguma dúvida? Não deixe de mandar um e-mail na lista do livro.

Capítulo 11

Validando os dados de entrada

Apesar de estar mais simples e já fazer boa parte de seu trabalho muito bem, o método `adiciona` ainda peca em um ponto muito importante. Os dados de entrada não são validados, permitindo que qualquer informação seja persistida no banco. Veja por exemplo o que acontece quando adicionamos um produto sem nenhum campo preenchido:

Fig. 11.1: Produto em branco adicionado na listagem.

O processo foi concluído sem que nenhum erro fosse apresentado, ou seja, um registro vazio foi adicionado no banco e ponto final. E não é só isso, tente por exemplo adicionar um produto com valor e quantidade negativa:

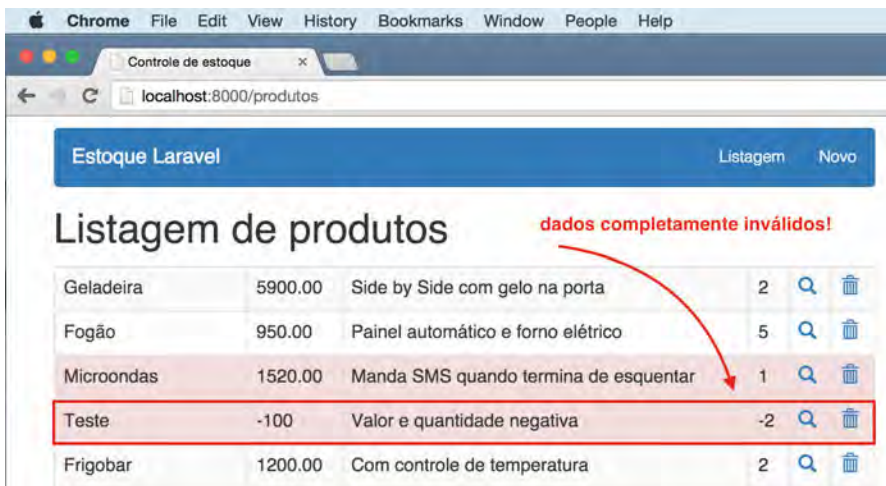



Fig. 11.2: Produto com valor e quantidade negativa.

Funciona, mas não deveria. Sempre que estamos adicionando recursos no banco de dados precisamos tomar esse cuidado, de validar os valores passados pelo usuário de acordo com a nossa regra de negócio. Em alguns casos, passar um valor negativo será totalmente válido, mas não é o nosso. O mesmo para as demais regras. Mas como validar cada valor que foi passado?

Claro, um `if` já resolveria o problema:

```
public function adiciona(){

  $valor = Request::input('valor');

  if(floatval($valor) < 0) {
    // volta para o formulário com uma mensagem de erro
  }

  // faz o restante do trabalho
}
```

Mas quantos desse seriam necessários para validar um formulário de 30 campos? Pense em como ficaria o método `adiciona`, cheio de regras de validação espalhadas nos diversos `ifs`. Um pesadelo de manutenção.

11.1 Validação com Laravel

Se é uma necessidade de muitos, com certeza o Laravel tem algo para nos ajudar. Esse é o espírito desse e dos outros frameworks também, facilitar a vida de quem desenvolve e oferecer recursos essenciais para nossos projetos.

Existem diversas formas de fazer validação dos dados, e o `Validator:make` é uma delas. Com ele você pode fazer algo como:

```
$validator = Validator::make(
    ['nome' => Request::input('nome')],
    ['nome' => 'required|min:5']
);
```

Repare que passamos dois arrays, um com os valores e outro com as regras de validação. Nesse exemplo, o nome será obrigatório (*required*) e precisará

ter no mínimo 5 caracteres. Você pode ver todas as regras de validações existentes neste link:

http://laravel.com/docs/validation#available-validation-rules

Agora, voltando ao exemplo, o método `make` retorna um `$validator` que tem uma série de métodos fundamentais para esse trabalho. Por exemplo, se tiver um erro de validação, preciso voltar para o formulário, certo? Podemos verificar isso com o método `fails`:

```
if ($validator->fails())
{
  return redirect()
    ->action('ProdutoController@novo');
}
```

Simples! O código do método `adiciona` completo ficará assim:

```
public function adiciona(){

    $validator = Validator::make(
        ['nome' => Request::input('nome')],
        ['nome' => 'required|min:5']
    );

    if ($validator->fails())
    {
      return redirect()
        ->action('ProdutoController@novo');
    }

    Produto::create(Request::all());

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Para testar, podemos abrir a página de adição de produtos e clicar em *adicionar*, sem preencher nada no campo `nome`. Veja que ele retornará para o formulário, sem persistir o produto no banco de dados.

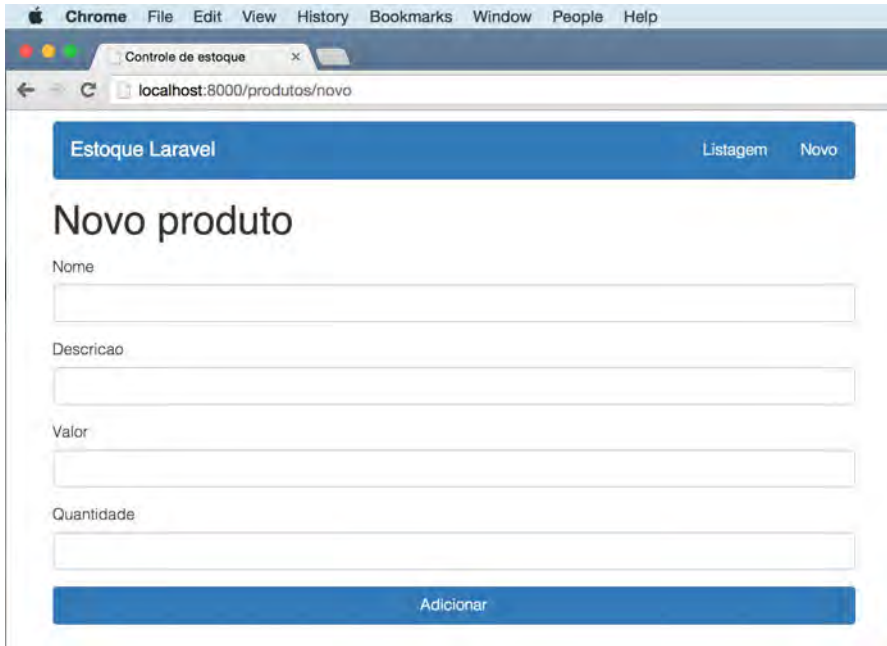



Fig. 11.3: Retornou ao formulário, porém sem exibir uma mensagem.

O problema é que dessa forma o usuário não sabe o que fez de errado, precisamos mostrar uma mensagem de feedback para ele! O `Validator` também tem um método que nos ajuda com isso, repare:

```
$messages = $validator->messages();
```

O método `messages` nos retorna um array com todas as mensagens de validação. Poderíamos, por exemplo, devolver essa lista de erros para a view. Mas, antes de nos preocuparmos com isso, existem outros detalhes que precisamos atacar.

O grande problema dessa abordagem é que, normalmente, queremos validar mais de um campo. Veja que apenas com a validação do `nome` o método `adiciona` já ficou enorme, imagine quando adicionarmos regras pra descrição, valor, quantidade e outros campos que podem vir a existir. Vai ficar enorme!

```
public function adiciona(){

    $validator = Validator::make(
     [
      'nome' => Request::input('nome'),
      'descricao' => Request::input('descricao'),
      'valor' => Request::input('valor'),
      'quantidade' => Request::input('quantidade')
     ],
     [
      'nome' => 'required|min:5'
      'descricao' => 'required|max:255',
      'valor' => 'required|numeric',
      'quantidade' => 'required|numeric'
     ]
    );

    if ($validator->fails())
    {
      return redirect()
        ->action('ProdutoController@novo');
    }

    Produto::create(Request::all());

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Isso porque nosso produto só tem 4 propriedades, imagine se tivesse 30! Seria um caos. Utilizar o `Validator` dessa forma é muito simples, mas costuma ser recomendado apenas para casos simples e isolados. Nos demais casos, recomenda-se a criação de uma nova classe, especialista por aplicar essas regras de validação.

11.2 Validando com Form Requests

Esse tipo de classe especial, cuja responsabilidade é validar os dados do formulário enviados pela requisição, é conhecida pelo Laravel como **Form Requests**. É uma classe normal, que você cria dentro do diretório `app/Http/Requests` e deve herdar da classe `Request` presente neste mesmo diretório.

Mas em vez de criar a classe na mão, adicionar os imports, namespace e todos esses detalhes, podemos deixar todo esse trabalho para o **Artisan**. Veja como é simples:

```
php artisan make:request ProdutosRequest
```

Muito parecido com quando estamos criando um novo modelo, mas em vez de `make:model`, faremos um `make:request`. Vamos testar? É só digitar esse comando no terminal, dentro da pasta do nosso projeto:

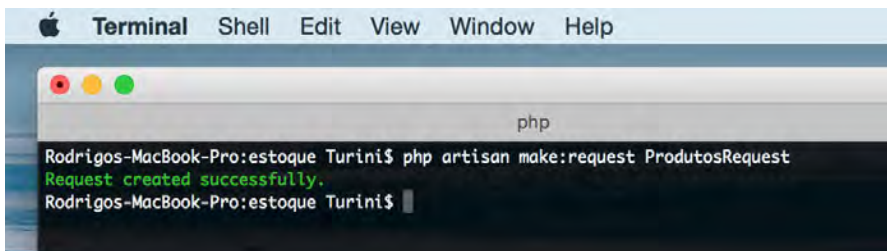



Fig. 11.4: Criando um form request com Artisan.

A mensagem **request created successfully** deve ser exibida e pronto, já podemos procurar pelo arquivo `ProdutosRequest` dentro de `app/Http/Requests`.

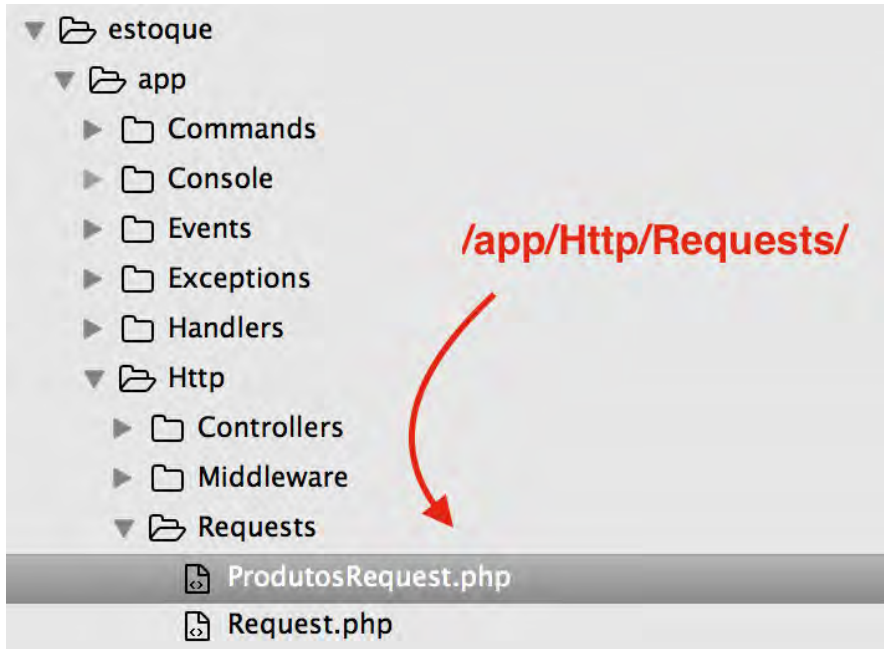



Fig. 11.5: ProdutosRequest criado na pasta app/Http/Requests.

Chamamos à classe de `ProdutosRequest` porque é uma prática comum utilizar esse sufixo `Request`, mas não é obrigatório. Você pode escolher o nome que preferir. No geral, a vantagem de seguir essa convenção é que apenas ao ler o nome da classe você já sabe do que se trata, é um **form request** de produtos!

A classe foi criada com a seguinte estrutura:

```
<?php namespace estoque\Http\Requests;

use estoque\Http\Requests\Request;

class ProdutosRequest extends Request {

    /**
     * Determine if the user is authorized
     * to make this request.
```

```
     *
     * @return bool
     */
    public function authorize()
    {
        return false;
    }

    /**
     * Get the validation rules that apply
     * to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            //
        ];
    }
}
```

Parece bem assustador, um bicho de sete cabeças! Mas no fim, você perceberá que ela é bem simples. Temos dois métodos, o primeiro, chamado `authorize`, é utilizado para informar se o usuário pode ou não fazer essa requisição.

Ele é executado antes mesmo da validação, portanto se o deixarmos como `false`, ninguém vai conseguir adicionar produtos. Normalmente nesse método podemos verificar se o usuário tem permissões, fazer consultas no banco de dados etc., mas em nosso caso, sempre queremos executá-lo, portanto basta mudar seu valor para `true`.

```
public function authorize()
{
    return true;
}
```

Já no outro método, o `rules`, retornamos um array com as regras de

validação, assim como estávamos fazendo no método `make` do `Validator`. Em nosso caso, será:

```
public function rules()
{
    return [
        'nome' => 'required|max:100',
        'descricao' => 'required|max:255',
        'valor' => 'required|numeric'
    ];
}
```

Com isso, o código completo da classe `ProdutosRequest` ficará assim:

```
<?php namespace estoque\Http\Requests;

use estoque\Http\Requests\Request;

class ProdutosRequest extends Request {

    /**
     * Determine if the user is authorized
     * to make this request.
     *
     * @return bool
     */
    public function authorize()
    {
        return true;
    }

    /**
     * Get the validation rules that apply
     * to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
```

```
            'nome' => 'required|max:100',
            'descricao' => 'required|max:255',
            'valor' => 'required|numeric'
        ];
    }
}
```

11.3 Usando Form Request ao adicionar

Agora que temos a classe especialista em fazer o trabalho, podemos tirar toda a complexidade das regras de validação de dentro do método do controller. Ele voltará a ficar assim:

```
public function adiciona(){

    Produto::create(Request::all());

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Mas de alguma forma precisamos ensinar ao Laravel que queremos aplicar as regras de validação do **form request** ao executar esse método e, para fazer isso, basta adicionar o `ProdutosRequest` como um argumento do método `adiciona`:

```
public function adiciona(ProdutosRequest $request){

    Produto::create(Request::all());

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Além disso, em vez de utilizar o `Request:all()` para pegar os parâmetros da requisição, usaremos o próprio **form request**. Veja que a mudança será mínima:

```
Produto::create($request->all());
```

O código do método `adiciona` ficará assim:

```
public function adiciona(ProdutosRequest $request){

    Produto::create($request->all());

    return redirect()
        ->action('ProdutoController@lista')
        ->withInput(Request::only('nome'));
}
```

Vale lembrar que precisamos adicionar o *import* da classe `ProdutosRequest` no início do nosso `ProdutoController`:

```
<?php namespace estoque\Http\Controllers;

use estoque\Produto;
use Request;
use estoque\Http\Requests\ProdutosRequest;

class ProdutoController extends Controller {
// restante do código
```

Vamos testar? Abrimos o formulário novamente e tentamos adicionar um produto sem preencher nenhum dado. O resultado será:

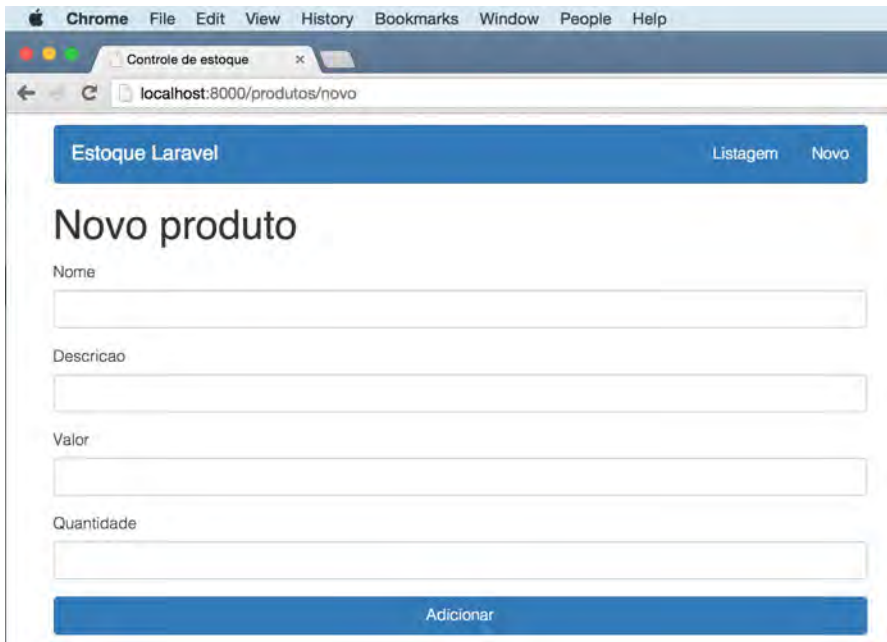



Fig. 11.6: Formulário de adição sem mensagem de validação.

Sem precisar configurar nada, automaticamente nossa requisição foi redirecionada de volta para o formulário! Claro, se você quiser ir pra outro lugar, bastaria adicionar um `if` com o método `fails` assim como fizemos anteriormente, mas por padrão, o **form request** já redireciona de volta para a página que fez a requisição. Legal, não é? Assim evitamos escrever aquele `if` chato.

Mas ainda não está perfeito, afinal, cadê a mensagem de validação? O usuário precisa de um feedback para saber o que fez de errado. Agora sim podemos atacar esse requisito.

11.4 Exibindo errors de validação

Exibir os erros é uma necessidade comum, e claro que o Laravel já pensou em uma forma de nos ajudar nisso. Você não precisa sempre adicionar as mensagens manualmente, depois de cada erro de validação. Isso é feito auto-

maticamente.

Uma variável chamada `errors` estará disponível na view após um ou mais erros de validação. Ela tem diversos métodos, por exemplo o `all`, que retorna um array com todas as mensagens. Para testar, adicione a seguinte instrução antes do seu formulário de adicionar produtos:

```
@foreach($errors->all() as $error)
    {{ $error }}
@endforeach
```

Agora basta tentar adicionar um produto vazio novamente:

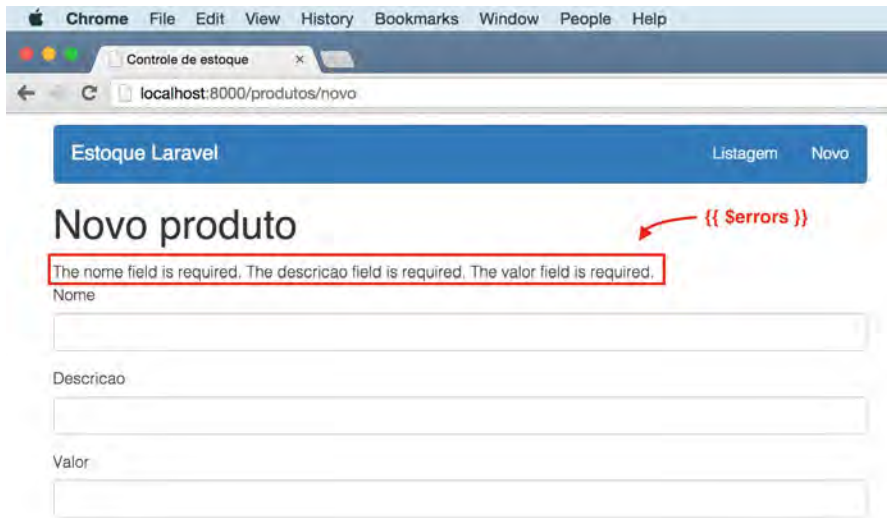



Fig. 11.7: Formulários com erros de validação.

.

Excelente! As mensagens de validação foram exibidas assim como esperado, mas ainda muito feio, sem nenhuma formatação e tudo na mesma linha. Podemos adicionar alguns estilos do bootstrap para deixar a aparência melhor:

```
<div class="alert alert-danger">
<ul>
```

```
  @foreach ($errors->all() as $error)
    <li>{{ $error }}</li>
  @endforeach
</ul>
</div>
```

O resultado está bem melhor!

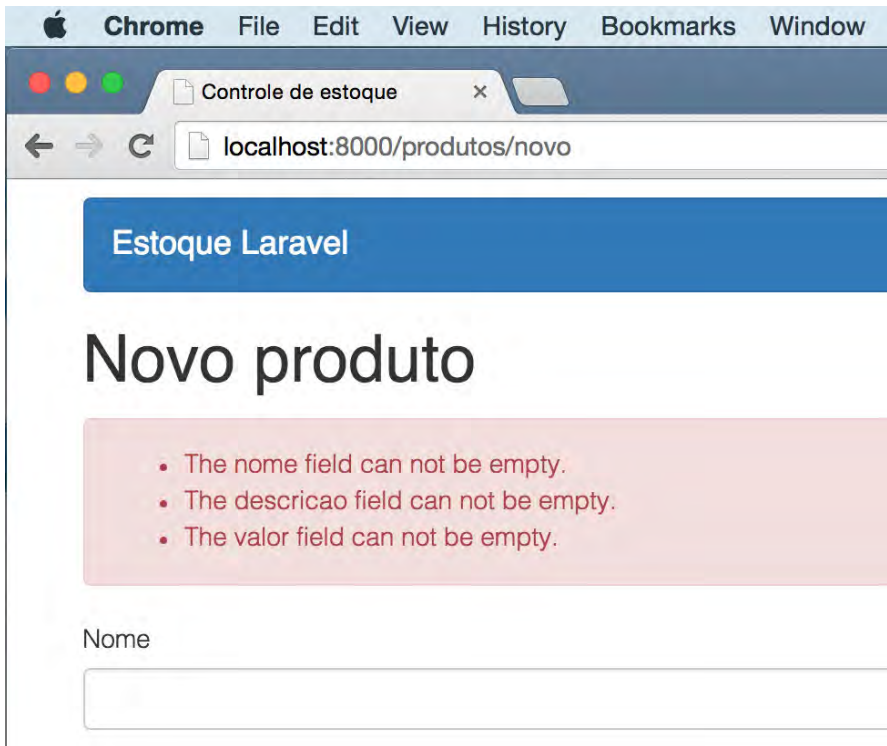



Fig. 11.8: Formulário com erros de validação.

Só falta um último detalhe, que pode ser percebido ao recarregar a view do formulário:

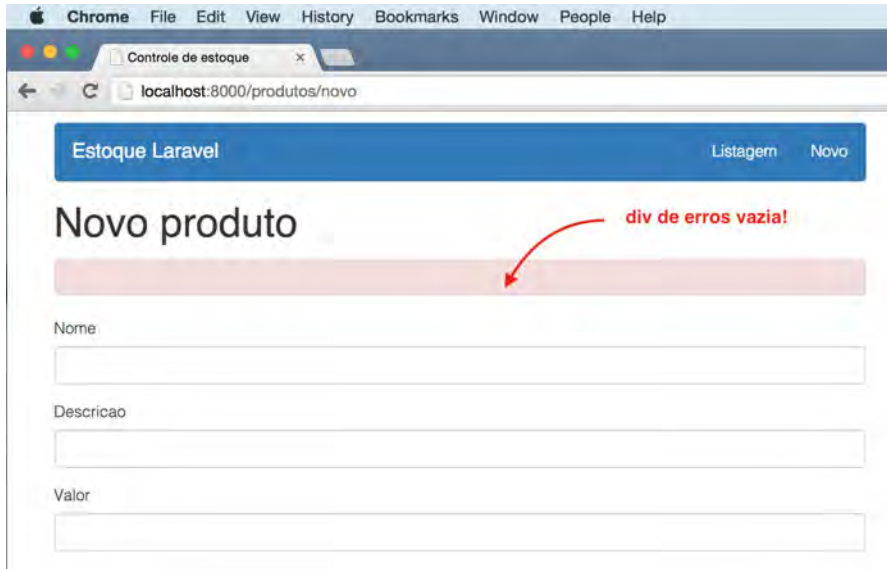



Fig. 11.9: Div de erros sempre presente.

Não queremos exibir a *div* de erros quando não existe nenhum erro de validação. Um simples `if` já resolverá esse problema:

```
@if (count($errors) > 0)
  <div class="alert alert-danger">
    <ul>
      @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif
```

Perfeito! O código final do arquivo `formulario.blade.php` deve ficar assim:

```
@extends('layout.principal')

@section('conteudo')
```

```
<h1>Novo produto</h1>

@if (count($errors) > 0)
  <div class="alert alert-danger">
    <ul>
      @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif

<form action="/produtos/adiciona" method="post">

  <input type="hidden"
      name="_token" value="{{{ csrf_token() }}}" />

  <div class="form-group">
    <label>Nome</label>
    <input name="nome" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Descricao</label>
    <input name="descricao" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Valor</label>
    <input name="valor" class="form-control"/>
  </div>
  <div class="form-group">
    <label>Quantidade</label>
    <input type="number"
        name="quantidade" class="form-control"/>
  </div>
  <button type="submit" class="btn btn-primary
      btn-block">Adicionar</button>
</form>
```

```
@stop
```

11.5 Customizando as mensagens

Outro ponto importantíssimo relacionado às mensagens de validação é a flexibilidade de customizá-las de acordo com nossa vontade ou necessidade. Por exemplo, e se em vez de mostrar *The nome field is required* eu quiser mostrar um outro texto, ou mesmo uma mensagem em português, onde devo alterar?

Existem diversas formas de fazer isso e, já que estamos usando **form requests**, adicionar um novo método chamado `messages` é uma delas. Quer ver como é simples? Vamos criar esse método dentro da classe `ProdutosRequest`. Ele deve retornar um array com a chave e valor que deve ser alterado. Por exemplo, se quisermos mudar a mensagem de *required* ele pode ficar assim:

```
public function messages()
{
  return [
     'required' => 'The nome field can not be empty.',
  ];
}
```

Veja que a mudança foi mínima, antes era *is required* (é obrigatório) e agora mudamos para *can not be empty* (não pode ser vazio). Claro, se quiser, também poderia usar uma mensagem em português. Vamos testar? É só abrir o formulário novamente e tentar adicionar um novo produto sem nenhuma informação preenchida.

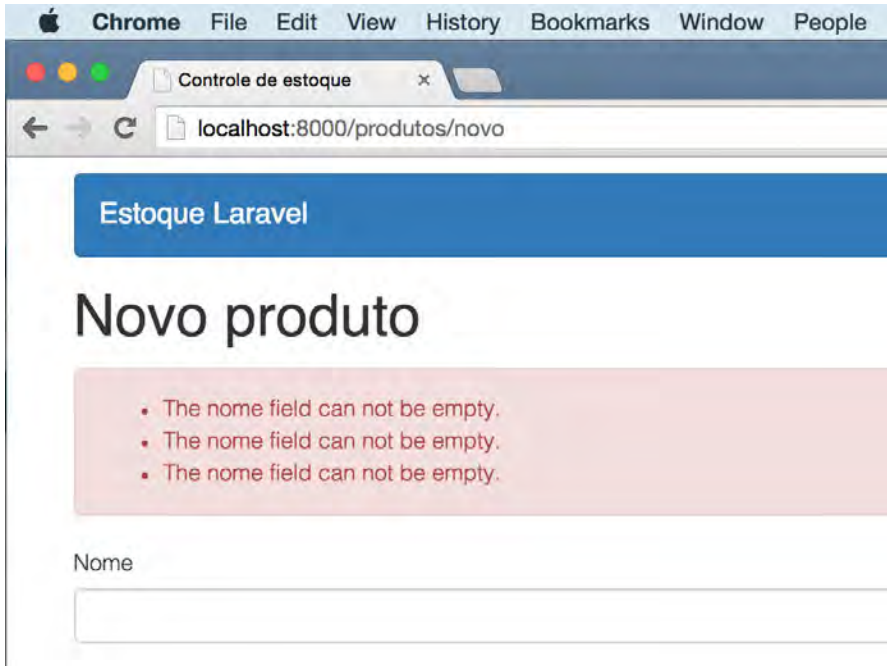



Fig. 11.10: Palavra 'nome' sempre aparecendo na mensagem de validação.

Ops, aconteceu algo inesperado. A nova mensagem foi exibida com sucesso, mas agora ficou com o campo `nome` fixo. Tente adicionar um produto sem descrição, por exemplo, a mensagem ainda estará com o texto `nome` no lugar de `descricao`. Como corrigir?

Em vez de utilizar o valor `nome`, experimente mudar para `:attribute` como a seguir:

```
public function messages()
{
  return [
     'required' => 'The :attribute field can not be empty.',
  ];
}
```

Essa variável especial será substituída pelo valor do campo que está sendo validado. Quer ver? Tentamos acessar um produto em branco novamente e o

resultado foi:

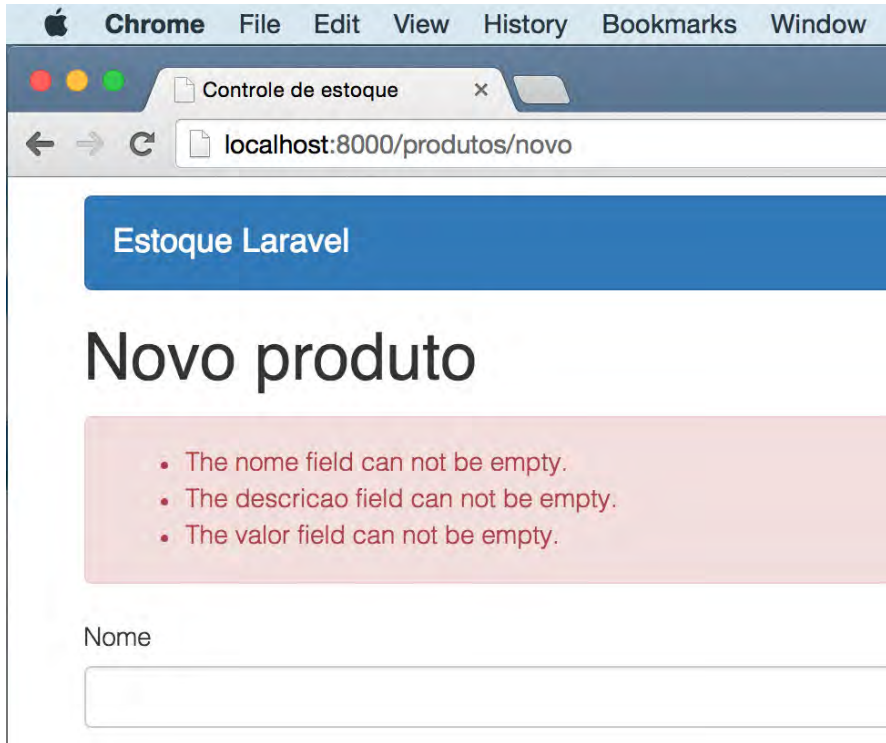



Fig. 11.11: Mensagem correta, com uso do *attribute*.

Agora sim, os valores `nome`, `descricao` e `valor` foram exibidos corretamente.

11.6 Customizando mensagens de um campo específico

Ao alterar a mensagem de *required*, dentro da classe `ProdutosRequest`, estamos aplicando isso para todos os campos obrigatórios desse formulário. Mas há momentos em que queremos customizar a mensagem apenas para um campo, como o `nome`. Para fazer isso, basta adicionar o prefixo `nome.` na chave da mensagem:

```
public function messages()
{
  return [
    'nome.required' => 'The :attribute field can not be empty.',
  ];
}
```

Agora essa regra será aplicada apenas para o nome!

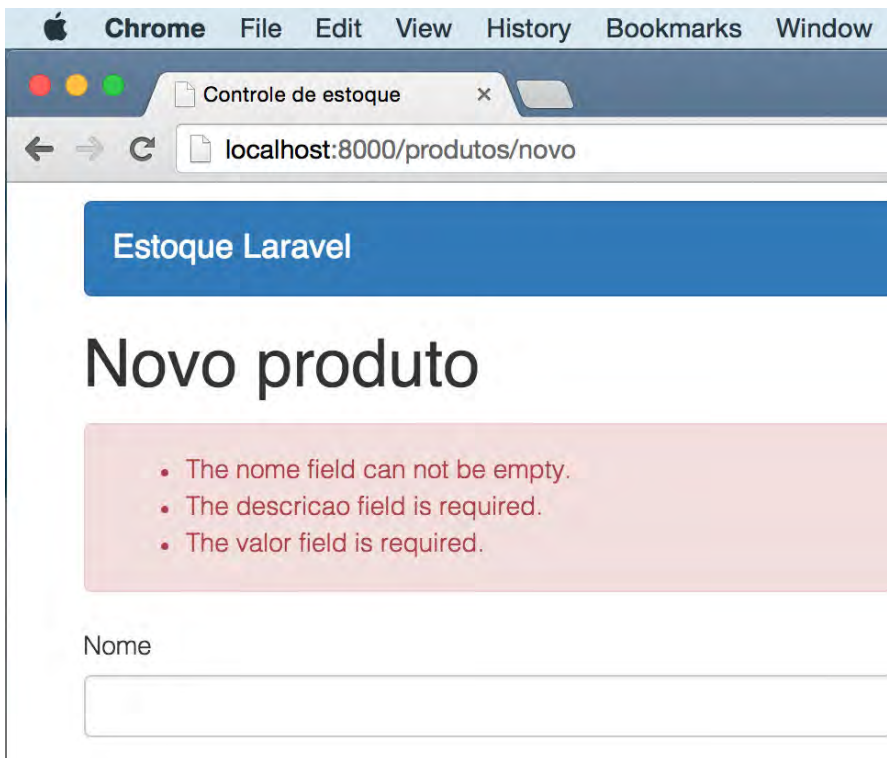



Fig. 11.12: Mensagem de validação do nome customizada.

Mas ainda tem uma questão, que pode ou não ser um problema. E se eu quiser aplicar essa regra para todos os campos de `nome` da aplicação? Da forma como fizemos, seria necessário repetir o código em cada **form request** que tivermos. No lugar disso, se você quiser aplicar a alteração para a aplica-

ção toda, você pode adicionar a mensagem no array `custom` definido dentro do `validation.php` da pasta `resources/lang/en`. Ele já estará assim:

```
'custom' => [
    'attribute-name' => [
        'rule-name' => 'custom-message',
    ],
],
```

Isso já nos dá uma dica de seu uso. Para o campo nome, por exemplo, o array ficaria assim:

```
'custom' => [
    'nome' => [
        'required' => 'The :attribute field can not be empty.',
    ],
],
```

Legal, não é? Claro, existem diversas outras formas de se validar e customizar as mensagens do formulário. Se quiser, você pode ler um pouco mais sobre validação nessa página da documentação do framework:

http://laravel.com/docs/validation

11.7 Boa prática: experiência de uso

O problema do nosso código atual é que, após submeter o formulário com erros de validação, o usuário será redirecionado para o formulário novamente, porém perderá todos os dados que digitou!

Quando temos poucos campos isso pode até passar despercebido, mas imagine como seria ruim ter que digitar novamente todos os valores de um formulário enorme toda vez que preencher um campo errado.

A solução para isso é simples e muitíssimo recomendada. Tudo o que precisamos fazer para manter os dados da requisição anterior que deu um erro de validação preenchidos é adicionar o método `old` no atributo `value` dos nossos *inputs*. Um exemplo seria:

```
<input name="nome" value="{{ old('nome') }}" />
```

Faremos o mesmo com cada um dos *inputs* do arquivo `formulario.blade.php`. Ao final, nosso HTML pode ficar assim:

```
@extends('layout.principal')

@section('conteudo')

<h1>Novo produto</h1>

@if (count($errors) > 0)
  <div class="alert alert-danger">
    <ul>
      @foreach ($errors->all() as $error)
        <li>{{ $error }}</li>
      @endforeach
    </ul>
  </div>
@endif

<form action="/produtos/adiciona" method="post">

  <input type="hidden"
      name="_token" value="{{{ csrf_token() }}}" />

  <div class="form-group">
    <label>Nome</label>
    <input name="nome"
        class="form-control" value="{{ old('nome') }}" />
  </div>
  <div class="form-group">
    <label>Descricao</label>
    <input name="descricao" class="form-control"
        value="{{ old('descricao') }}"/>
  </div>
  <div class="form-group">
    <label>Valor</label>
    <input name="valor" class="form-control"
        value="{{ old('valor') }}"/>
  </div>
```

```
  <div class="form-group">
    <label>Quantidade</label>
    <input type="number" name="quantidade"
        class="form-control" value="{{ old('quantidade') }}"/>
  </div>
  <button type="submit" class="btn
      btn-primary btn-block">Adicionar</button>
</form>

@stop
```

Capítulo 12

Autenticação e segurança

Um ponto fundamental que ainda não exploramos no livro é quanto à autenticação e segurança de nossa aplicação. Da forma que está, qualquer um visualiza, cria ou apaga produtos, mas nem sempre podemos permitir que isso aconteça. Veremos nesse capítulo como o Laravel nos ajuda com esse trabalho, com uso de seus famosos `middlewares` e mecanismo de `Auth`.

12.1 Tela de login e autenticação

A grande verdade é que tudo está pronto, em uma aplicação real você provavelmente precisará mudar um ou outro detalhe, mas boa parte do código será totalmente reutilizada em seus projetos.

Para testar, precisaremos voltar aquelas linhas que apagamos do arquivo `routes.php` no primeiro capítulo. Não se preocupe se não lembrar, o importante é saber que quando você cria uma aplicação o seguinte conteúdo já

vem registrado nos arquivo de rotas:

```
Route::get('home', 'HomeController@index');

Route::controllers([
    'auth' => 'Auth\AuthController',
    'password' => 'Auth\PasswordController',
]);
```

Copie novamente para lá. Seu `routes.php` pode ficar assim:

```
Route::get('/produtos', 'ProdutoController@lista');
// outras rotas omitidas

Route::get('home', 'HomeController@index');

Route::controllers([
    'auth' => 'Auth\AuthController',
    'password' => 'Auth\PasswordController',
]);
```

Ótimo! Veja que há uma rota `/home` apontando para o método `index` do `HomeController`, que também deve ter sido criado automaticamente com o projeto. Caso esse `controller` não exista em seu projeto, você pode criá-lo manualmente com o seguinte conteúdo:

```
<?php namespace estoque\Http\Controllers;

class HomeController extends Controller {

    public function __construct()
    {
        $this->middleware('auth');
    }

    public function index()
    {
        return view('home');
    }
}
```

Não se preocupe em entender o construtor dessa classe, pelo menos por enquanto. Em breve seu uso ficará bem claro. Além disso, veja que o método `index` retorna uma `view` chamada `home`. Crie também esse arquivo, chamado `home.blade.php`, com o seguinte conteúdo:

```
@extends('app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-10 col-md-offset-1">
            <div class="panel panel-default">
                <div class="panel-heading">Home</div>

                <div class="panel-body">
                    You are logged in!
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Vamos ver o que tem lá dentro? É só acessar: http://localhost:8000/home.

Talvez para grande surpresa, o resultado ao tentar acessar essa URL será um redirecionamento para a tela de login. Isso mesmo, essa tela já existe e, por sinal, tudo que existe nela já funciona!

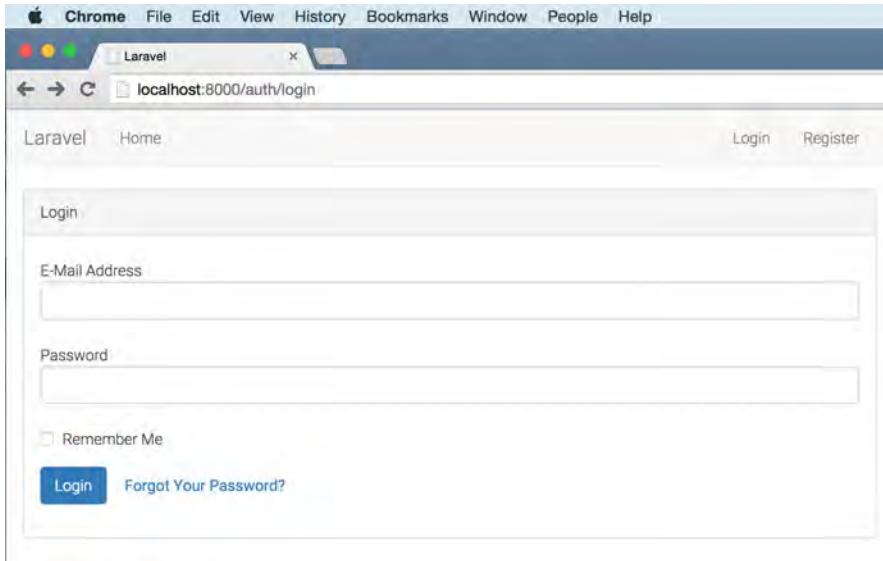
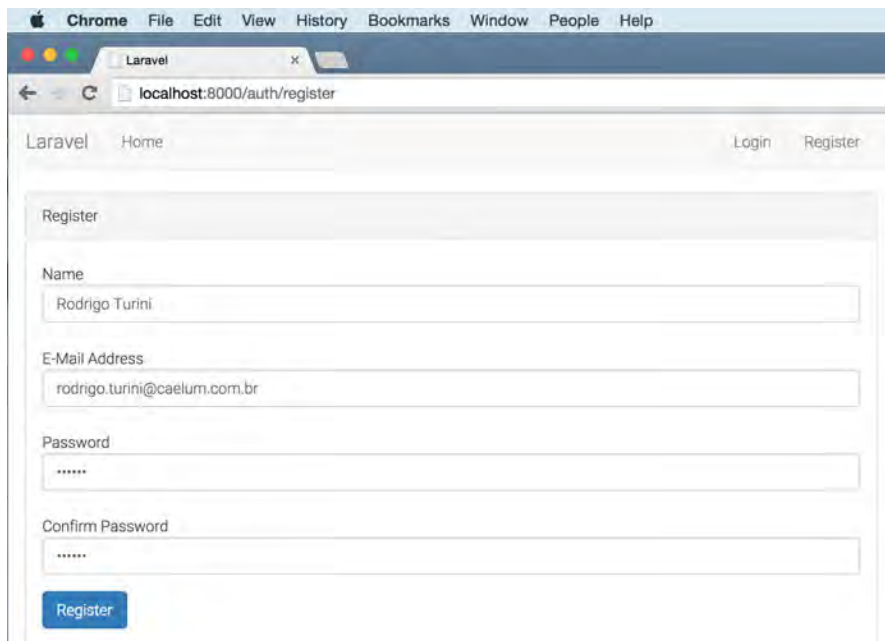



Fig. 12.1: View com formulário de login.

Veja que, além do formulário de login, ela oferece uma opção de reset de senha e registro de novos usuários. Vamos testar? Como não temos nenhum usuário cadastrado, podemos criar um novo registro clicando na opção `Register` do canto direito do menu.

Fig. 12.2: Formulário de registro de usuários.

Preencha o formulário e clique no botão de submit para ver o resultado. O usuário será criado, e você será redirecionado para a tela `/home`, que estava protegida por login. Logo veremos essa parte de segurança, como limitar acesso aos recursos da aplicação e tudo mais, mas por enquanto, vamos entrar um pouco mais a fundo nessa funcionalidade de login.

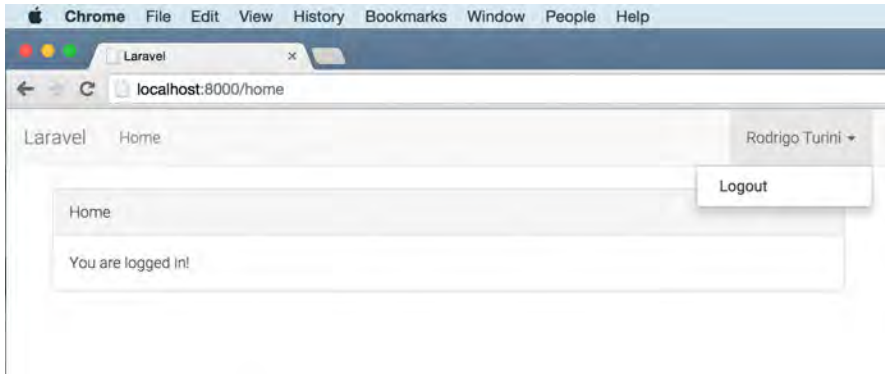



Fig. 12.3: View padrão da URL /home.

Repare que o nome de seu usuário estará disponível no menu principal e, ao ser clicado, exibirá a opção de logout. Mas onde tudo isso está definido? Como foi criado? Como modificar?

12.2 Entendendo o login da aplicação

Vou responder essas perguntas por partes, ok? Primeiro vamos entender onde essas *views* foram definidas e alguns detalhes de sua implementação.

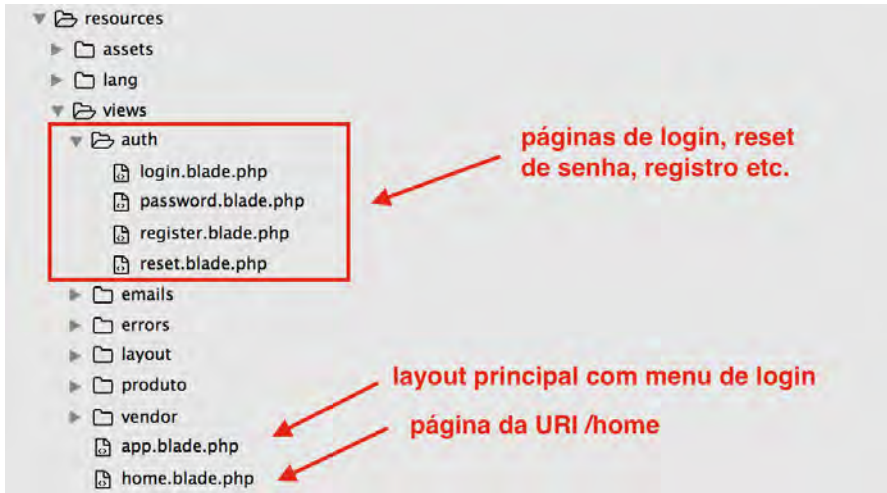



Fig. 12.4: Diretório com arquivos de autenticação.

Veja que a pasta `resources/views/auth` mantém todas as páginas relacionadas a essa lógica de autenticação. Podemos sim apagar tudo e recriar todas as views, formulários, validação etc. Mas e no próximo projeto? Apagamos e fazemos tudo de novo? Vai ser tudo muito parecido. Vale lembrar que a grande sacada do framework é essa, oferecer o que existe em comum entre todos os projetos, pra você gastar seu tempo no que realmente importa.

Você pode e deve editar, adicionar novos campos e fazer qualquer adaptação que for necessário de acordo com as necessidades do seu projeto, mas com certeza isso será mais fácil agora que tudo já está pronto.

Veja que a página `home.blade.php` é bem simples, tem apenas um painel que mostra a mensagem de que o usuário está logado:

```
@extends('app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-10 col-md-offset-1">
            <div class="panel panel-default">
                <div class="panel-heading">Home</div>
```

```
                <div class="panel-body">
                    You are logged in!
                </div>
            </div>
        </div>
    </div>
</div>
@endsection
```

Um detalhe que quero que vejam nesse arquivo é que ele herda de `app`, que é o layout principal que já vem pronto no Laravel. Essa é a cereja do bolo, o lugar onde o mecanismo de autenticação do framework começa a aparecer. Abra o arquivo `app.blade.php` e veja que ele é muito parecido com nosso layout principal, o `principal.blade.php`. A grande diferença está no `navbar`, que faz um `if` verificando se o usuário está logado ou não:

```
<ul class="nav navbar-nav navbar-right">
  @if (Auth::guest())
    <li><a href="/auth/login">Login</a></li>
    <li><a href="/auth/register">Register</a></li>
  @else
    <li class="dropdown">
      <a href="#" class="dropdown-toggle"
          data-toggle="dropdown" role="button"
              aria-expanded="false">
          {{ Auth::user()->name }}
          <span class="caret"></span>
      </a>
      <ul class="dropdown-menu" role="menu">
        <li><a href="/auth/logout">Logout</a></li>
      </ul>
    </li>
  @endif
</ul>
```

Há dois pontos importantíssimos nesse trecho de HTML e, em ambos, a interface `Auth` foi utilizada. Essa interface (*facade*) é quem nos ajuda com boa parte do trabalho de autenticação. Vamos entender como ela funciona?

Veja que no `if` o método `guest` (convidado) é verificado. Esse método, como você já deve imaginar, determina se o usuário já está logado na aplicação. Ele é bastante útil em diferentes camadas do nosso projeto: Na view, para determinar se o nome do usuário ou link de login deve ser exibido, no controller ou qualquer camada intermediária de autorização para verificar se o usuário pode ou não acessar o conteúdo etc.

Outro método interessante utilizado por essa view é o `user`, que retorna uma instância do usuário logado na aplicação. Repare que ele é utilizado para exibir o nome do nosso usuário no seguinte trecho do código:

```
{{ Auth::user()->name }}
```

Legal! Mas e quanto ao código de autenticação em si, usado para logar ou deslogar um usuário? Bem, em vez de apenas ficar na teoria, que tal criarmos o nosso próprio sistema de login só por diversão? O código será mais simples do que parece.

12.3 Criando uma lógica de autenticação

Para começar, podemos criar um novo controller onde ficará toda essa lógica de autenticação. Vamos chamá-lo de `LoginController` e, lembrando, ele deverá ficar dentro de `app/Http/Controllers`. Antes de criar o arquivo e digitar tudo, que tal dessa vez usar o `artisan`? Assim como fizemos pra criar o modelo, também podemos usar o comando `make:controller`.

```
php artisan make:controller LoginController --plain
```

Esse `--plain` que passamos como argumento significa que não queremos que ele crie nenhum método. Deve ser um controller em branco. Sem essa opção, ele adicionará uma série de métodos comuns para as operações básicas de adicionar, listar, remover etc. O famoso *CRUD*.

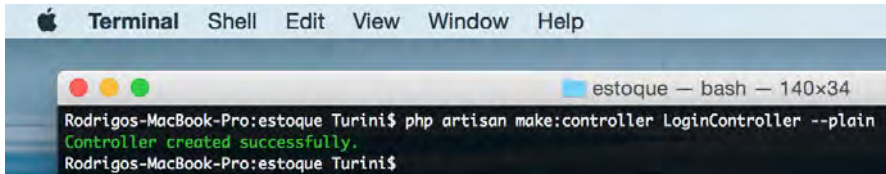



Fig. 12.5: Criando controller com Artisan.

O resultado será uma mensagem de sucesso e pronto, podemos abrir o arquivo `LoginController` que já estará assim:

```
<?php namespace estoque\Http\Controllers;

use estoque\Http\Requests;
use estoque\Http\Controllers\Controller;

use Illuminate\Http\Request;

class LoginController extends Controller {

    //
}
```

Vamos agora criar um novo método, chamado `login`:

```
public function login()
{
    // código vai aqui
}
```

Em um caso tradicional criaríamos também um método que envia para a view de formulário, que faria um `POST` com os dados para essa ação de `login`, mas como estamos apenas testando e não queremos reinventar a roda , passaremos o usuário e senha direto pela URL.

No arquivo de rotas, vamos ensinar ao Laravel que ao acessar a URI `/login` nosso método de login deverá ser chamado:

```
Route::get('/login', 'LoginController@login');
```

Quase tudo pronto, já podemos começar o código de autenticação. No método `login`, queremos pegar o `email` e `password`, verificar se ele é valido e, em caso de sucesso, retornar uma mensagem com o nome do usuário logado; caso contrário, um erro. Em resumo, o código ficará assim:

```
public function login()
{
    $credenciais = Request::only('email', 'password');

    if( /*credenciais são válidas*/ ) {
        return "Usuário NOME logado com sucesso";
    }

    return "As credencias não são válidas";
}
```

Mas como saber se o usuário existe em nosso banco de dados? Poderíamos sim escrever um SQL que procura o usuário por e-mail e senha, mas o `Auth` já cuida desse trabalho para não termos que nos preocupar nem com isso. Quer ver como é simples? Basta passar o `array` com e-mail e senha para seu método `attempt`:

```
public function login()
{
    $credenciais = Request::only('email', 'password');

    if(Auth::attempt($credenciais)) {
        return "Usuário NOME logado com sucesso";
    }

    return "As credencias não são válidas";
}
```

Além de e-mail e senha, você pode passar qualquer informação válida do usuário como parâmetro no array de credenciais.

Vale lembrar que para usar o `Auth` dentro do controller precisaremos fazer o *import*. A classe `LoginController` completa deve ficar assim:

```
<?php namespace estoque\Http\Controllers;

use estoque\Http\Requests;
use estoque\Http\Controllers\Controller;

use Auth;
use Request;

class LoginController extends Controller {

    public function login()
    {
        $credenciais = Request::only('email', 'password');

        if(Auth::attempt($credenciais)) {
            return "Usuário NOME logado com sucesso";
        }

        return "As credencias não são válidas";
    }
}
```

Vamos testar? Primeiro tente acessar a lógica de login com um usuário inválido. Um exemplo seria: http://localhost:8000/login?email=teste@teste.com.br&password=123456

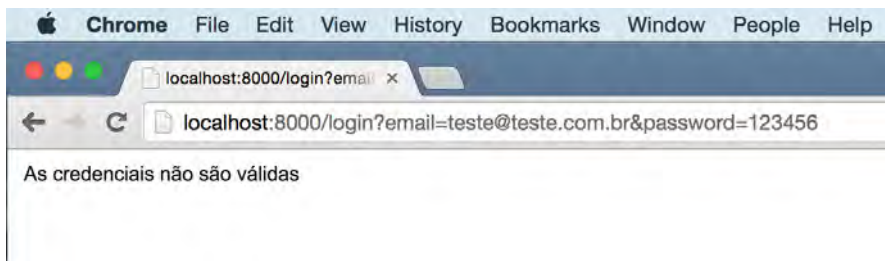



Fig. 12.6: Mensagem de credencias inválidas.

A mensagem **As credencias não são válidas** foi exibida como esperado. Experimente agora passar o login e senha que você registrou ante-

riormente. Em meu caso, será: http://localhost:8000/login?email=rodrigo.turini@caelum.com.br&password=123456

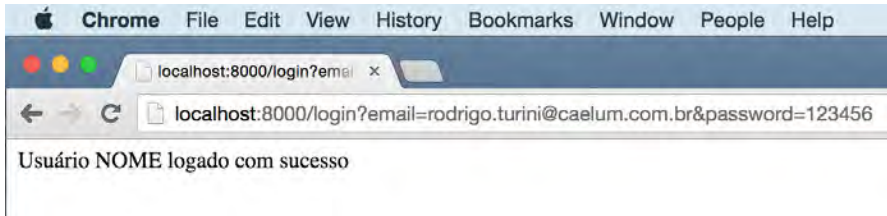



Fig. 12.7: Mensagem de sucesso, porém com a palavra 'nome' fixo.

Sucesso! Mas no lugar de `NOME`, queremos mostrar o nome do usuário autenticado. Lembra como se faz? É só utilizar o método `user` e acessar a propriedade que você quiser. O código ficará de seguinte forma:

```
if(Auth::attempt($credenciais)) {
    return "Usuário ".
        Auth::user()->name
        ." logado com sucesso";
}
```

Agora sim, ao acessar a URL novamente o resultado será:

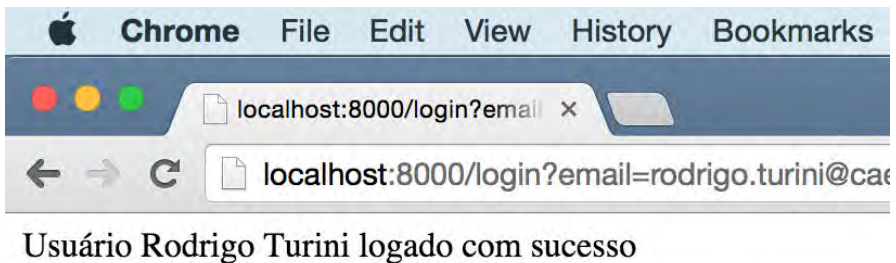



Fig. 12.8: Mensagem de confirmação de login.

Há mais um monte de recursos interessantes no `Auth`. Por exemplo, se você quiser usar a função "lembrar a senha" na sua aplicação, você pode passar

um `boolean` como segundo argumento para o método `attempt`. O usuário vai ficar autenticado por tempo indefinido ou até alguém manualmente fazer o `logout`.

```
if(Auth::attempt($credenciais, true)) {
    //...
}
```

Por sinal, fazer o `logout` também é bem simples! Repare:

```
Auth::logout();
```

Há ainda uma série de métodos que podem e com certeza serão úteis. Alguns deles são:

```
// verifica apenas se as credenciais são
// válidas, sem necessariamente se logar
Auth::validate($credentials);

// para logar um usuário de determinado id
Auth::loginUsingId(7);

// para ver se o usuário está logado
Auth::check();

// ou então, verificar o próprio usuário
Auth::user();
```

Legal, não é? Mas vale lembrar de que a parte de autenticação já está toda pronta, dificilmente vamos usar todos esses métodos.

Agora que já conhecemos um pouco de como o Laravel cuida da autenticação, vamos entrar em outro ponto muito importante: a autorização. Não basta estar logado, é necessário estabelecer o que o usuário pode ou não acessar.

Se quiser, aproveite para explorar um pouco mais antes de prosseguir. Que tal mudar o template das páginas de autenticação para usar o `principal`? Além disso, você pode adicionar o `if` nesse layout para verificar se o usuário está logado ou não, mostrando o nome ou os links de login assim como o

app.blade.php faz. Fique à vontade para fazer essas e outras alterações no sistema, é uma excelente oportunidade para relembrar alguns recursos que vimos até aqui.

12.4 Autorização com Middlewares

A questão do login foi explorada, sabemos que isso já está pronto, mas mesmo sem estarem logados os usuários da nossa aplicação ainda conseguem visualizar, adicionar e excluir produtos. Não queremos que isso aconteça, esses devem ser privilégios exclusivos para quem está autenticado.

No início do capítulo, quando acessamos a URI /home pela primeira vez, fomos redirecionados para uma tela de login. Isso aconteceu porque, de alguma forma, foi configurado que para acessar essa lógica era necessário estar autenticado no projeto. Queremos o mesmo, mas onde configurar?

Poderíamos fazer isso de diversas formas. Um exemplo seria validando no próprio método, como no exemplo a seguir do método remove:

```
public function remove($id){
    if (Auth:guest())
    {
        return redirect('/auth/login');
    }
    $produto = Produto::find($id);
    $produto->delete();
    return redirect()->action('ProdutoController@lista');
}
```

Veja que um simples if resolve o problema. Ao tentar remover um produto sem estar logado, o usuário será redirecionado para a tela de login. É exatamente o que precisamos! Porém, imagine só a quantidade de vezes que vamos precisar repetir esse if? Muitas. E quanto maior for o projeto, mas ifs repetidos. Seria inviável.

É aí que entra o **Middleware**. Ele funciona como um filtro, por onde passam todas as requisições HTTP antes de chegar em nossos controllers. O nome pode parecer intimidador, mas não há nada de muito complicado. Quer ver? Antes de sair usando os *Middleware*s existentes, vamos criar nossa própria regra de autenticação.

Em vez de criar a classe manualmente, podemos usar o comando `make:middleware` do Artisan para fazer esse trabalho por nós. Veja como é simples:

```
php artisan make:middleware AutorizacaoMiddleware
```

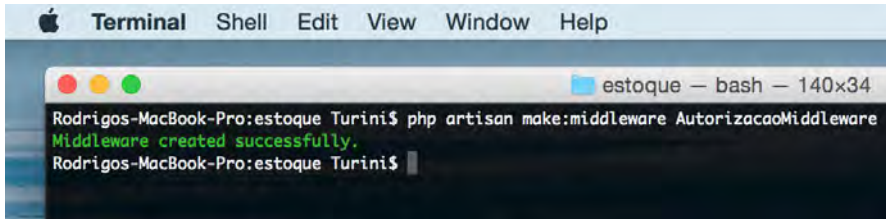



Fig. 12.9: Middleware de autorização[[

Agora basta abrir o arquivo `AutorizacaoMiddleware` dentro do diretório `app/Http/Middleware` e a seguinte estrutura estará pronta:

```
<?php namespace estoque\Http\Middleware;

use Closure;

class AutorizacaoMiddleware {

    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

}
```

Todo o trabalho acontece nesse método chamado `handle`, que é executado sempre que uma nova requisição é feita em nossa aplicação. Repare que ele recebe como argumento a `$request` com os dados da requisição atual e também uma `Closure` chamada `$next`. Esse segundo argumento é o responsável por determinar ou não se nosso controller será chamado. Se o método `handle` não retornar `$next($request)` da forma que está, a aplicação ficará parada nesse ponto.

De forma prática, podemos fazer algo como:

```
public function handle($request, Closure $next)
{
    if(/* não pode acessar */) {
        return redirect('/auth/login');
    }
    return $next($request);
}
```

Veja que, nesse caso, quando o usuário não tiver acesso a uma determinada lógica ou URI, redirecionamos para a página de autenticação. Caso contrário, retornamos o `$next($request)` que continuará com a execução do nosso código normalmente.

Simples, não é? Mas o que colocar nesse `if`? A resposta é: depende da nossa necessidade. Você pode fazer desde consultas no banco de dados para verificar o privilégio do usuário e de acordo com ele permitir o acesso ou não, até um simples `Auth:guest` para verificar se o usuário está logado, que é justamente o que precisamos neste momento.

O código ficará assim:

```
public function handle($request, Closure $next)
{
    if(\Auth::guest()) {
        return redirect('/auth/login');
    }
    return $next($request);
}
```

Excelente, mas agora que o *middleware* está criado, todas as requisições vão passar por ele? Nós somos quem decidimos. Se quisermos fazer isso,

bastaria adicionar o nome dessa nossa classe no atributo `$middleware` de uma classe chamada `Kernel`, presente no diretório `app/Http`.

Ao abrir essa classe, ela estará parecida com:

```
<?php namespace estoque\Http;

class Kernel extends HttpKernel {

    /**
     * The application's global HTTP middleware stack.
     *
     * @var array
     */
    protected $middleware = [
        // outros registros omitidos
        'Illuminate\Session\Middleware\StartSession',
        'Illuminate\View\Middleware\ShareErrorsFromSession',
        'estoque\Http\Middleware\VerifyCsrfToken',
        'estoque\Http\Middleware\AutorizacaoMiddleware',
    ];

    /**
     * The application's route middleware.
     *
     * @var array
     */
    protected $routeMiddleware = [
        'auth' => 'estoque\Http\Middleware\Authenticate',
        'auth.basic' => 'Illuminate\Auth\Middleware
            \AuthenticateWithBasicAuth',
        'guest' => 'estoque\Http\Middleware
            \RedirectIfAuthenticated',
    ];
}
```

Observe que essa classe possui dois arrays, chamados `$middleware` e `$routeMiddleware`. O primeiro é utilizado sempre que quisermos que um *middleware* seja executado **em todas as requisições**. Bastaria adicionar nossa classe nesse array e pronto, mas isso seria um grande problema.

O nosso código sempre verifica se o usuário está logado e, caso não esteja, redireciona para a tela de login. Como **toda requisição** passará pelo *middleware*, ele novamente verificará se está logado, e redirecionará outra vez para a tela de login. O código entrará em *looping* de redirect.

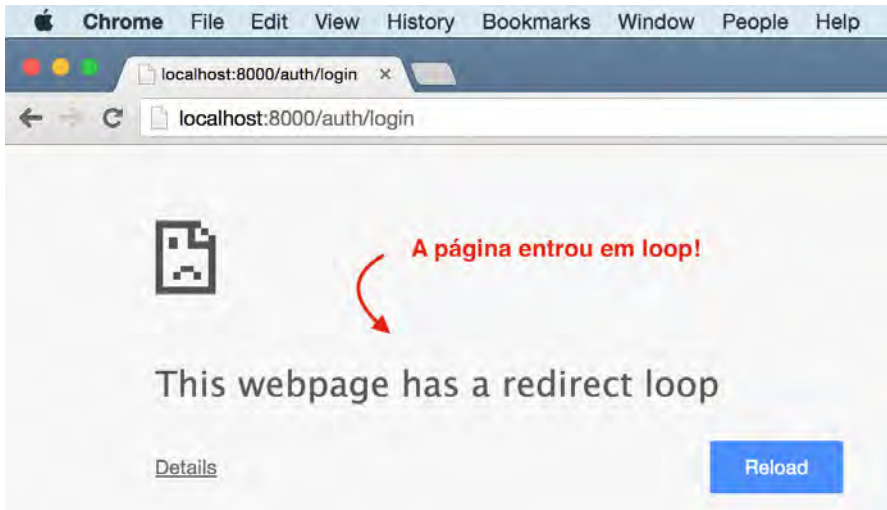



Fig. 12.10: Página em redirect loop.

E agora? A solução é simples, bastaria adicionar uma condição no `if` que verifique que a `request` acessada não é a de `auth/login`:

```
public function handle($request, Closure $next)
{
    if(!$request->is('auth/login') && \Auth::guest()) {
        return redirect('/auth/login');
    }
    return $next($request);
}
```

Excelente, problema do `loop` resolvido. Faça as alterações para testar, ao acessar qualquer URL enquanto estiver deslogado, você será redirecionado para a tela de login.

12.5 Registrando o middleware para rotas específicas

Imagine agora que queremos bloquear apenas algumas rotas, como a de adicionar e remover produto, mas as demais devem ser liberadas. O usuário não precisará estar logado para acessar a listagem, por exemplo.

Da mesma forma como fizemos com a URI de login, podemos ir adicionando condições para liberar as páginas de listagem, detalhes do produto ou qualquer outra que quisermos, mas o problema dessa solução é que a todo momento que uma nova página que não deve ser bloqueada for criada, precisaremos lembrar de adicionar uma nova condição no *middleware*. Além disso, o código ficaria bem feito, cheio de `ifs` ou com um único `if` enorme!

No lugar disso, o Laravel nos oferece a possibilidade de ativar um *middleware* para rotas específicas. Lembra que a classe `Kernel` tinha dois arrays, o `middleware` e o `$routeMiddleware`? Pois bem, esse segundo atributo serve justamente para isso.

Vamos remover nossa classe do array `middleware` e, em vez disso, adicioná-la ao `$routeMiddleware` como a seguir:

```
protected $routeMiddleware = [
    'auth' => 'estoque\Http\Middleware
        \Authenticate',

    'auth.basic' => 'Illuminate\Auth\Middleware
        \AuthenticateWithBasicAuth',

    'guest' => 'estoque\Http\Middleware
        \RedirectIfAuthenticated',

    'nosso-middleware' => 'estoque\Http
        \Middleware\AutorizacaoMiddleware',
];
```

Veja que já existem outros *middlewares* registrados nesse array também, logo falaremos sobre eles. O importante agora é perceber que nesse caso associamos a chave `nosso-middleware` ao nosso `AutorizacaoMiddleware`.

Associando middlewares com a rota

Agora que já está registrado, precisamos associar o `nosso-middleware` às rotas que devem ser filtradas. Existem diversas formas de fazer isso, e uma delas é pelo arquivo `routes.php`.

```
Route::get('/produtos/remove/{id}', [
    'middleware' => 'nosso-middleware',
    'uses' => 'ProdutoController@remove'
]);
```

O problema dessa abordagem é que além de poluir bastante o nosso arquivo de rotas, ainda nos obriga a repetir essas linhas do *middleware* em cada uma delas.

Associando middlewares ao controller

Em vez disso, podemos partir para um caminho bem mais interessante, que é registrar o *middleware* no construtor do nosso controller. Veja como é simples:

```
<?php namespace estoque\Http\Controllers;

use estoque\Produto;
use Request;
use estoque\Http\Requests\ProdutosRequest;

class ProdutoController extends Controller {

    public function __construct()
    {
        $this->middleware('nosso-middleware');
    }

    // restante do código omitido
}
```

Mas ainda não é o que queremos, pois isso fará com que todos os métodos do controller passem pelo filtro do *middleware*, ou seja, ao tentar acessar a listagem novamente seremos redirecionados para a tela de login.

Para evitar isso, podemos passar como segundo argumento apenas os métodos em que queremos aplicar esse *middleware*:

```
public function __construct()
{
    $this->middleware('nosso-middleware',
        ['only' => ['adiciona', 'remove']]);
}
```

Neste caso, apenas o método `adiciona` e `remove` serão filtrados, todos os demais funcionarão normalmente mesmo sem o usuário estar logado. Além do `only`, você também pode utilizar a chave `except` para dizer quais métodos não devem ser filtrados.

Vamos testar as alterações? Primeiro acessando a listagem de produtos. Tudo deve funcionar:

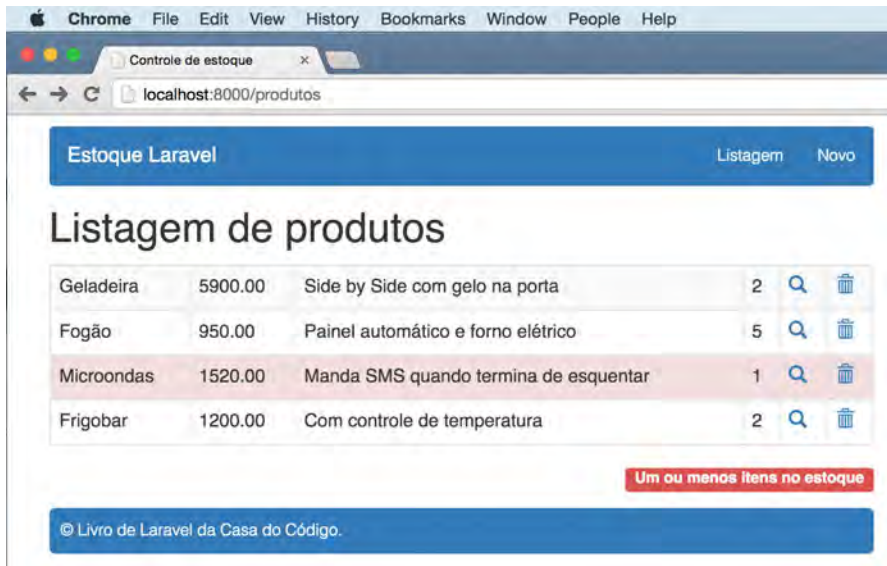
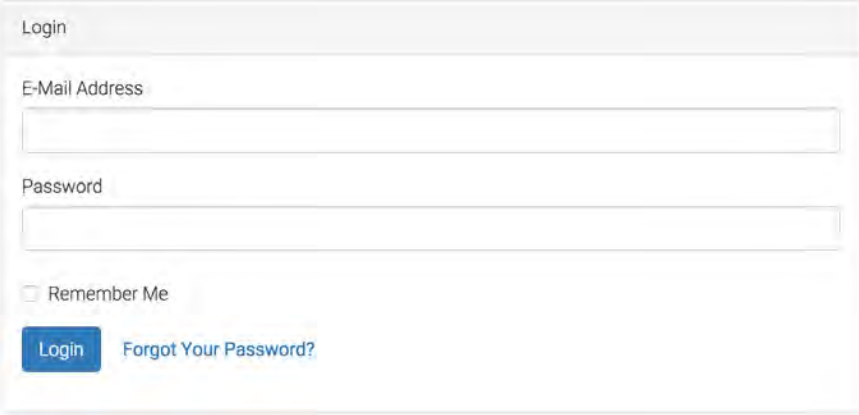



Fig. 12.11: Listagem de produtos.

Agora tente remover um dos produtos:

Fig. 12.12: Página de login.

Sucesso! Fomos redirecionados para a tela de login.

12.6 Utilizando o middleware padrão

Vimos como criar e registrar nosso próprio *middleware*, afinal, é um recurso importantíssimo. Mas, na realidade, para um caso simples como esse o código todo já está pronto, portanto não precisaríamos desse trabalho todo.

Experimente mudar o construtor do controller para que utilize um *middleware* chamado `auth`:

```
<?php namespace estoque\Http\Controllers;

use estoque\Produto;
use Request;
use estoque\Http\Requests\ProdutosRequest;

class ProdutoController extends Controller {

    public function __construct()
    {
        $this->middleware('auth',
```

```
            ['only' => ['adiciona', 'remove']]);
    }
    // restante do código omitido
}
```

O efeito será o mesmo. Legal, não é? O Laravel nos entrega de mão beijada toda a parte de autenticação e autorização.

Capítulo 13

Mais produtividade com Artisan

Durante o livro, vimos um recurso muito atrativo do Laravel, o `Artisan`. Essa ferramenta de linha de comando nos torna ainda mais produtivos, pois cuida de boa parte do código de *boilerplate*. Ao criar um novo modelo, por exemplo, precisamos lembrar de adicionar o *namespace*, dizer que ele herda de `Model`, adicionar o import para a *superclasse* etc. Muito trabalho, não é? E é sempre a mesma coisa, só muda o nome da classe. É aí que o **Artisan** entra, um simples comando e pronto, toda essa rotina será feita para nós.

Mas a ferramenta não para por aí. Além de criar modelos, controllers, form requests, entre outros, ela também nos oferece diversos recursos como o `serve`, utilizado para rodar a aplicação no servidor de desenvolvimento do Laravel. Muito conveniente! O **Artisan** é um verdadeiro cinto de utilidades do programador produtivo.

13.1 Como lembrar de todos os comandos?

É fundamental ter em mente que quanto mais comandos você conhecer, mais proveito você vai tirar da ferramenta. Mas, levando em consideração que existem diversos, como seria possível decorar todos? A resposta é simples: não decore! O único comando de que você precisa se lembrar é o `list`, que retorna uma lista com todas as opções disponíveis. Sabendo ler e interpretar a saída desse comando, você consegue usar qualquer outro.

Vamos pôr as mãos na massa? Abra o terminal e, de dentro da pasta de seu projeto, execute o comando:

```
php artisan list
```

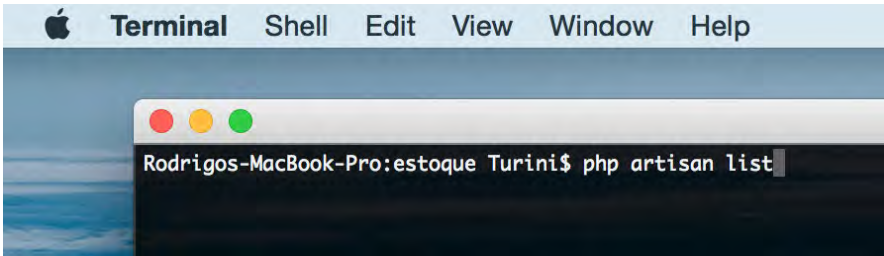



Fig. 13.1: Listagem de comandos do Artisan.

Uma lista nem um pouco pequena de comandos disponíveis será exibida. Aqui vai um resumo sobre os principais deles e também alguns exemplos práticos. Você pode e deve ir testando todos os comandos que quiser conhecer melhor.

- **down**: coloca a aplicação em modo de manutenção. Para testar, experimente rodar `php artisan down` e acessar qualquer URL da sua aplicação. O resultado será parecido com:

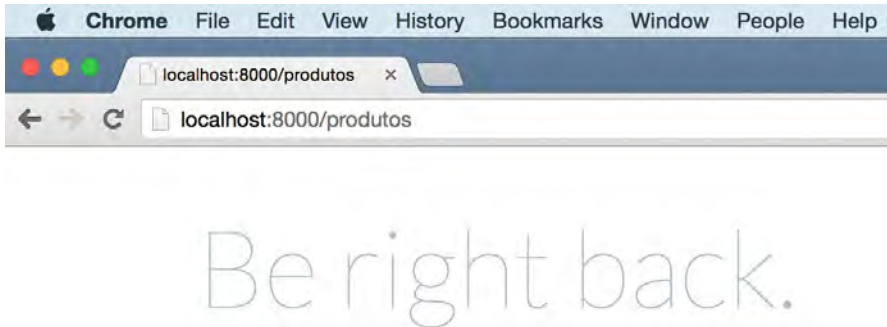



Fig. 13.2: Aplicação do modo de manutenção.

- **up**: tira a aplicação do modo de manutenção. Se você fez o teste anterior, basta rodar `php artisan up` para que tudo volte ao normal.

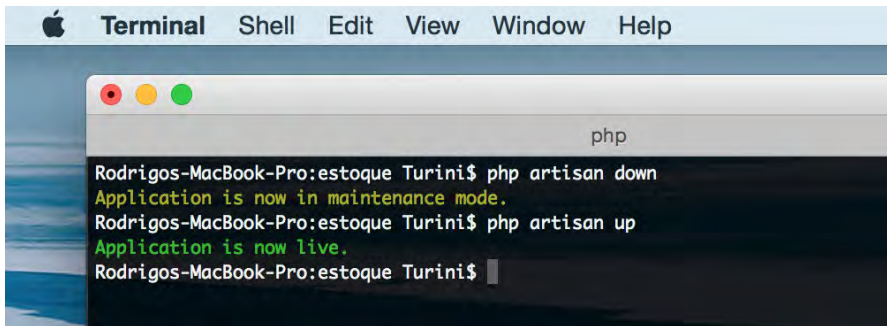
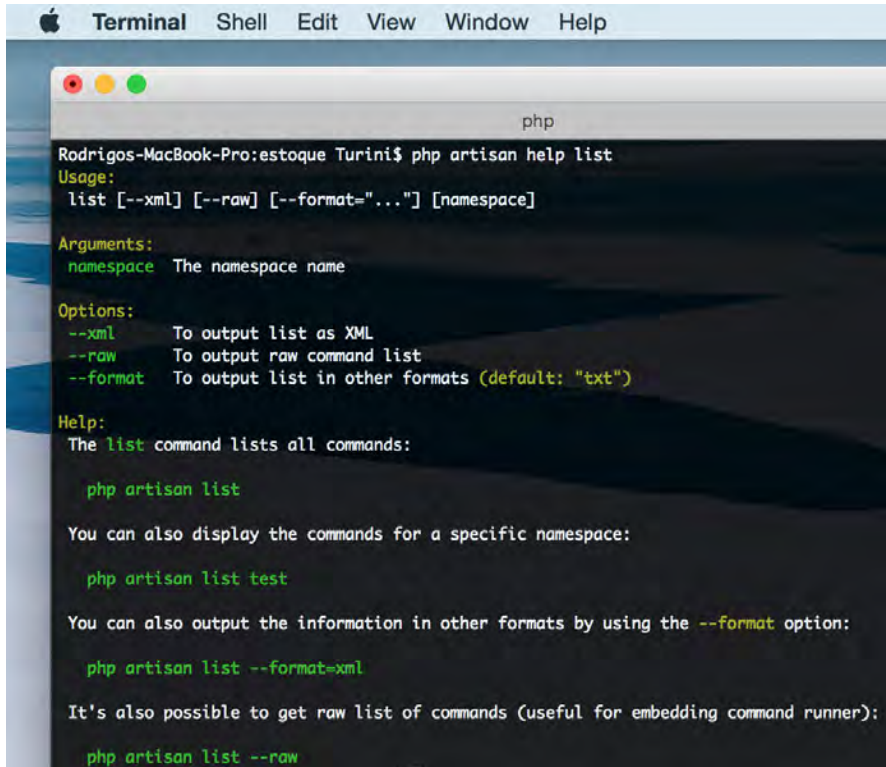



Fig. 13.3: Comandos down e up pelo terminal.

- **help**: exibe informações de ajuda para um comando. Por exemplo, se eu quero saber como o comando `php artisan list` funciona, basta digitar `php artisan help list`. Os possíveis argumentos, opções e alguns exemplos serão exibidos como a seguir:

Fig. 13.4: Informações de ajuda do comando list.

- **tinker**: interage com sua aplicação. Esse comando é muito útil e poderoso. É possível executar querys no banco de dados, utilizar qualquer classe do projeto etc.

```
Rodrigos-MacBook-Pro:estoque Turini$ php artisan tinker
Psy Shell v0.3.5 (PHP 5.5.21 — cli) by Justin Hileman
>>> DB::select('select * from produtos');
=> [
       <stdClass #00000000312fa188000000014713f21f> {
           id: 1,
           nome: "Geladeira",
           valor: "5900.00",
           descricao: "Side by Side com gelo na porta",
           quantidade: 2
       },
       <stdClass #00000000312fa189000000014713f21f> {
           id: 2,
           nome: "Fogão",
           valor: "950.00",
           descricao: "Painel automático e forno elétrico",
           quantidade: 5
       },
```

Fig. 13.5: Acessando dados do banco pelo Artisan Tinker.

- **list**: lista todos os comandos.
- **optimize**: otimiza a performance do framework.
- **serve**: sobe o servidor de desenvolvimento do PHP.
- **app:name**: adiciona o namespace da aplicação.

Os comandos a seguir, que inclusive já utilizamos, têm como propósito a criação de classes.

- **make:console**: cria um novo comando do Artisan.
- **make:controller**: cria um novo controller.
- **make:middleware**: cria um novo middleware.
- **make:model**: cria um novo modelo do Eloquent.

- **make:request**: cria um novo form request.

Há ainda opções interessantes para controle de cache e otimizações.

- **cache:clear**: limpa o cache da aplicação.
- **config:cache**: cria um arquivo de cache deixando a consulta de configurações mais rápida.
- **config:clear**: remove o arquivo de cache de configurações.

Por fim, alguns que nos ajudam com rotas:

- **route:cache**: cria um arquivo de cache para deixar o registro de rotas mais rápido.
- **route:clear**: remove o arquivo de cache de rotas.
- **route:list**: lista todas as rotas registradas. Este comando é especialmente útil! Experimente executar `php artisan route:list` para ver o resultado. Além de exibir as URIs, os métodos HTTP, actions e middleware associados a eles serão detalhados.

Esses são apenas alguns dos muitos comandos existentes, aproveite para explorar todas essas funcionalidades. Conhecer recursos que os frameworks nos oferecem é um grande diferencial, pois torna nosso dia a dia muito mais produtivo e divertido.

Capítulo 14

Os próximos passos

Existem diversas formas para você continuar seus estudos. Eu recomendo fortemente que você exercite todos os códigos escritos nos capítulos desse livro, que podem ser encontrados no repositório:

https://github.com/Turini/estoque-laravel

Para criar uma intimidade ainda maior com o framework, experimente passear por sua documentação e testar cenários além dos aqui sugeridos. Coloque a mão na massa. A partir de agora, descobrir novos recursos, utilizar novos frameworks etc. passa a ser seu trabalho do dia a dia.

Bons estudos!